THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR	Alynn B. Klassen
TITLE OF THESIS	S*(QM-1): AN EXPERIMENTAL EVALUATION OF THE HIGH LEVEL MICROPROGRAMMING LANGUAGE SCHEMA S* USING THE NANODATA QM-1

DEGREE FOR WHICH THESIS WAS PRESENTED MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED 1981

THE UNIVERSITY OF ALBERTA

S*(QM-1): AN EXPERIMENTAL EVALUATION OF THE HIGH LEVEL MICROPROGRAMMING LANGUAGE SCHEMA S* USING THE NANODATA QM-1

by

© Alynn B. Klassen

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

IN

COMPUTING SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

1981

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled S*(QM-1): AN EXPERIMENTAL EVALUATION OF THE HIGH LEVEL MICROPROGRAMMING LANGUAGE SCHEMA S* USING THE NANODATA QM-1 submitted by Alynn B. Klassen in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE in COMPUTING SCIENCE.

Abstract

The high level language schema S* is a partially specified, machine-independent microprogramming language which is used as a tool in the development of machine-dependent languages. S* is *instantiated* into S*(M1), for a particular microprogrammable machine M1, by completing the language specification based on the architectural features of M1. The use of S* is introduced in the context of the family of architectural design and implementation languages [S*], using as an example the development of a language directed architecture.

A high level language S*(QM-1) for *nanoprogramming* the Nanodata QM-1 is instantiated to determine the usefulness and viability of S* based on actual experiences. During this instantiation we also focused our attention on the specification and underlying philosophy of S* and investigated the process of instantiation.

We conclude that S*, with several minor changes and additions, is indeed a valuable tool for the development of high level microprogramming languages. The parallel and sequential flow of control constructs play an especially important role in ensuring correct programs in S*(QM-1). The adoption of a new construct **parbegin** is recommended for use in specifying a series of data independent statements which may be executed in any order allowing for the most efficient production of code possible.

Acknowledgements

I wish to express my deep appreciation to Dr. Subrata Dasgupta for his guidance, intellectual stimulation and editorial comments during the research and writing of this thesis. I would also like to thank Dr. John Tartar for his support during the initial research stage.

Special thanks are extended to the other members of the QM-1 User's Group, namely Steven Sutphen, Marius Olafsson and Douglas Rideout for their comments and criticisms during the development of S*(QM-1). Thanks also to Leo Hartman and Chris Grey for their comments on the final draft of the thesis.

Finally, I would like to thank my wife, Angela, for her patience and encouragement during all phases of this work.

Table of Contents

List of Figures

Chapter 1
Introduction

The development of a computer architecture from its inception to final implementation is an intellectually difficult, time consuming and arduous task. To meet these challenges and produce a result which is correct, acceptable, and on time, the designer must rely on as many tools as possible. In the initial planning stages an overall design methodology should be selected which encourages communication and employs a hierarchical top-down approach with smooth transitions from the specification phase through progressively more complex levels. One such methodology, presently under development, is the family of design and implementation languages, [S*] [DASG81a].

One member of the [S*] family is the high level microprogramming language schema, S*, which is to be the focus of this thesis. S* was first presented by Dasgupta [DASG78] and:

> *... is a* **partially specified** *(and therefore, partially machine independent) language such that for a given host machine M1, a particular language S*(M1) obtains when M1's properties are used to complete the specifications of S*. We say that S* is*

instantiated into S*(M1) with respect to M1.[1]

As the above quote intimates S* is not a panacea presenting us with the long awaited universal microprogramming language but attacks the problem of language development from a different direction.

In [DASG78] three ground rules are established upon which a microprogramming language should be based:

a. *The ability to construct control structures for designating clearly, and without ambiguity, both sequential and parallel flow of control.*

b. *The ability to describe and name arbitrarily, microprogrammable data objects or parts of such data objects.*

c. *The ability to construct microprograms whose structure and correctness can be determined and understood without reference to any control store organizations.*

With these points in mind S* provides essentially a framework employing many programming constructs found in the high level block structured language PASCAL. These constructs have been added to, and tailored to meet the needs and requirements of the horizontal microprogramming environment where timing, data dependencies and resource

---

[1] Subrata Dasgupta, "Some Implications of Programming Methodology for Microprogramming Language Design", *Microprogramming, Firmware and Restructurable Hardware*, ed. G. Chroust and J. Mulbacker, (Amsterdam: N-H, 1980), p. 244.

conflicts play a significant role.

Since its inception the language schema S* has, except for a partial example instantiation for the VARIAN 75 [DASG78], remained untried and untested. The objective of this thesis is the evaluation of S* using a threefold approach:

1. To examine the specification and underlying philosophy of the high level microprogramming language schema S*.
2. To investigate the process of instantiation and tools which may aid in this procedure.
3. To draw conclusions about the usefulness and viability of S* based upon experiences gained from performing an actual instantiation.

The Nanodata QM-1 [NANO79] was chosen as the experimental vehicle because it is widely recognized, with its 360-bit wide horizontal instruction word and use of **residual control**, to be one of the most difficult user-microprogrammable machines to program. With the successful development of an instantiated language S*(QM-1), it would be reasonable to expect that the process of instantiation would be no harder for most other machines.

In addition, the benefit of a high level language for algorithm design and implementation would be invaluable for users of the QM-1 who up to this time have had to program in a low level *nanoassembler* language. The difficulty of learning and using this language goes a long way in

explaining why only one major project [DEMC76] has been completed since the QM-1 was purchased in 1973 by the Department of Computing Science at the University of Alberta.

To solve the problems involved with the development of a high level *nanoprogramming* language a QM-1 Architecture Group headed by Dasgupta[2] was established in September 1980. Two avenues of attack were chosen one of which was the design of a language and compiler. The other tackled the thorny problem of microcode compaction [RIDE81a] which deals with the detection of parallelism between microoperations which are then packed in the fewest number of horizontal words while at the same time remaining conflict free. Both endeavors have been progressing relatively autonomously with the only interaction being the development of an intermediate tuple-oriented language.

1.1 Thesis Organization

A review of high level microprogramming languages is presented to provide the reader with a perspective of the language schema S* with respect to related work in this field. The second chapter introduces S* by placing it in the context of the family of design languages [S*] and illustrates its use in the overall design and implementation of an application-directed architecture. This is followed by

[2]Other members included Steven Sutphen, Marius Olafsson, Douglas Rideout, and the author.

an examination of the process of instantiation to shed light on the general approach to language design based on S*. Next the description of S* constructs is considered and based on this review and our experience with S*(QM-1) some initial recommendations for changes in S* are made.

Chapter three introduces the QM-1 and supplies the necessary architectural background so that language design decisions involving the instantiation of S*(QM-1) may be understood. The more important factors affecting the instantiation are presented in the fifth chapter. S* is expanded by adding several new constructs and a language which is solely procedure oriented is obtained, supporting the designer's objective for the QM-1 architecture. A complete description of the syntax and semantics of S*(QM-1) is included in Appendix 1. Chapter six details the design of a compiler and associated preprocessor, and discusses the communication between the parsing and semantic analysis phase, and the compaction phase.

The final chapter discusses conclusions and recommendations about S*, the process of instantiation, S*(QM-1) and the QM-1. The QM-1 is included to show how slight modifications in its design could provide more flexibility in its use.

1.2 Background

The adaptation of high-level language constructs, structured programming and related concepts for microprogram development have been under active investigation since 1970. The motive behind this work is clear--these constructs and programming methods reduce the chance of errors through well structured programs which are easier to understand and modify. The programmer is freed from the more intricate details of coding, allowing a less complex transformation of algorithms into programs, resulting in a more reliable product.

Most researchers have been content to use constructs and methods employed in existing languages such as PL/1, ALGOL 60, and PASCAL. With the exception of S*, little attention has been paid to implementing new constructs specific to microprogramming to supplement those in existing languages. Rather, researchers use these languages as a ready-made vehicle to support more complex fields of study. Three broad categories can be identified: detection of parallelism and optimization of microcode (efficiency), machine independence of microprograms (portability), and formal verification of microprograms (correctness). We shall briefly review five languages found in the literature and show how they have contributed to this field of study.

The first attempt at a high level language for microprogramming was MPL by Eckhouse [ECKH71]. The language was designed around PL/1 and addressed the problem of

machine independent programs for compilation into a vertical microinstruction format. It employed procedures with local and global variables based on six data types from register (one dimensional arrays) through events (machine testable conditions) to constants. A limited set of constructs--**if**...**then**...(**else**), **goto** and assignment--are used; looping is accomplished using a conditional statement. The language is now of historical interest only as the difficulties associated with horizontal microinstructions, with several microoperations executing concurrently, soon became the center of attention.

SIMPL [RAMA74] was the first language to address horizontal microinstruction formats with the objective of writing machine independent programs using a sequential statement specification. A *single identity principle* based on the multiprocessing concept of the *single assignment property* was developed which ignored the ordering of statements and relied solely on data dependencies to perform microcode optimization. Constructs found in ALGOL 60 were used: **if**...**then**...**else**, **while**... **do**, **for**...**do**, and **case** statements for multiway branching. The controversy surrounding the **goto** was avoided by not including it in the language. No data structuring was available, variables were identified only with registers and the set of operators was fixed.

Three languages, two of which emerged from doctoral theses, followed the introduction of SIMPL. One, STRUM

[PATT76], was devised to provide a method for the formal verification of well structured microprograms. The language was machine dependent and implemented on the Burroughs D-Machine. A rich set of constructs derived from PASCAL were used: **if...then...else**, **case**, **select**, **while...do**, **repeat...until**, **for...do**, **loop...pool**, macros and procedures. Arithmetic operations were limited to those that could be accomplished in one pass through an ALU or shifter. The designer concluded that formal verification coupled with well structured programming was a realistic method for providing reliable, efficient microprograms.

The final two, EMPL [DEWI76a, DEWI76b] and S*, address the question of efficiency and portability. Both investigators developed algorithms for the detection of parallelism from a sequential specification and proposed methods to handle portability.

EMPL's approach to portability is via the *extensibility* concept, such as is found in ALGOL 68, where a completely specified *core* language is extended by a programmer to *customize* it for use with a particular target machine. New data types, and operations which may be performed on them, are created to take advantage of particular micro-operations, such as a hardware instruction decoding unit, not directly supported by the core language. Portability was assured by compilers, for machines without this direct support, by supplying code to implement the missing feature. The core language has limited constructs

(which may not be added to) and permitted only one core type, the integer and vectors of integers.

Dasgupta's approach was to implement two new parallel constructs **cobegin...coend**, and **dur...do...end** based on his work in developing algorithms for the detection of parallelism [DASG76] in sequentially specified programs. These would, for the first time, allow programmers the ability to explicitly specify parallelism within their programs. With these constructs in mind he developed a *language schema*, S*, using data structures (the most powerful seen to date in microprogramming) and constructs from PASCAL which provide a partially specified "core" language. The language designer completes the syntactic and semantic definition of S* for a particular machine based on it's idiosyncrasies. Thus there would be a different language, but similar in its constructs, for each different microprogrammable machine.

Chapter 2

S* and the Process of Instantiation

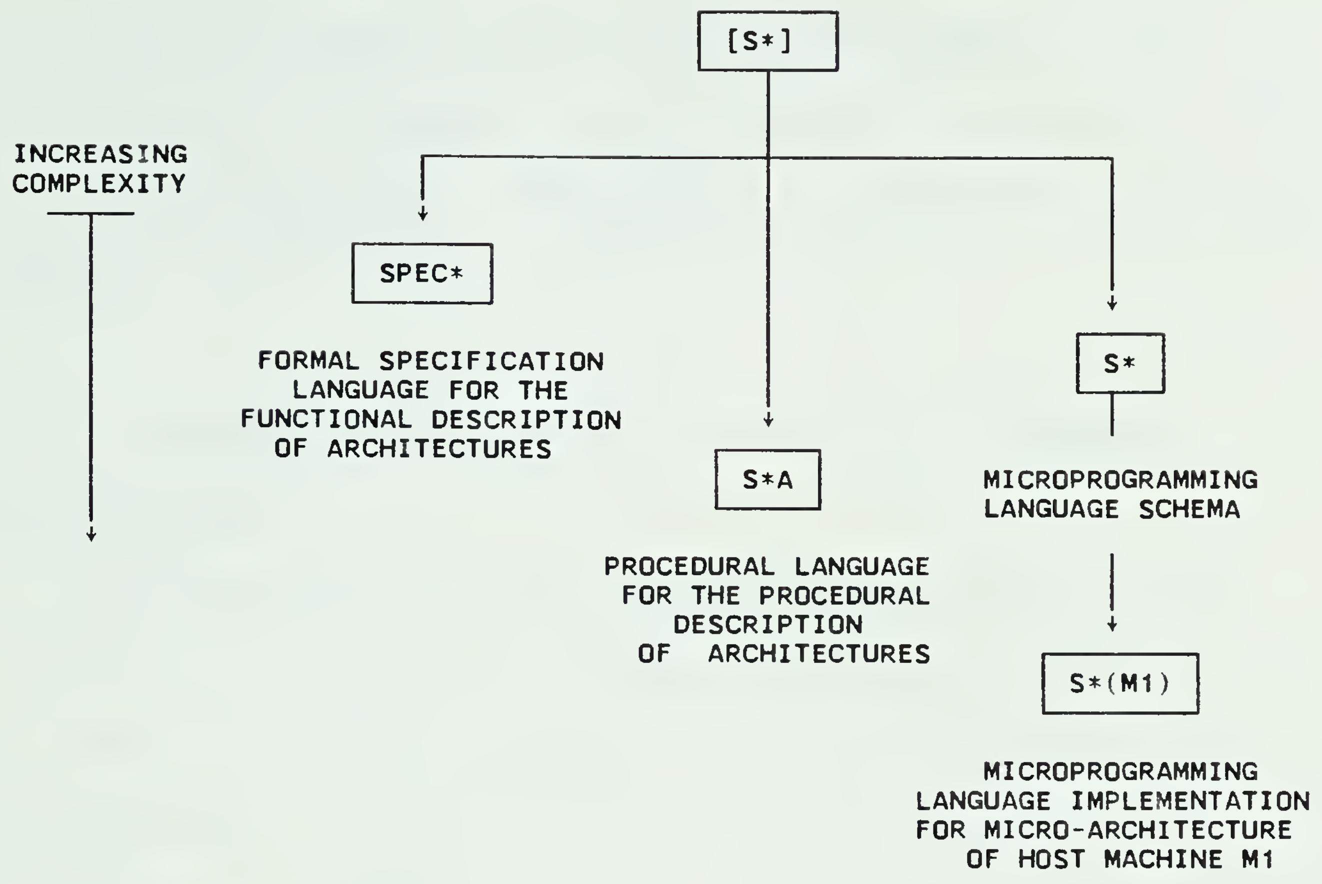
2.1 A Design Methodology

2.1.1 [S*] Family of Languages

[S*] is a family of design and implementation languages for the description and realization of computer architectures based on a unified top-down approach. It presently incorporates a three tier procedure where each lower level represents an increase in complexity with which the designer is faced. The transition from one level to the next is meant to be as smooth and natural as possible and is accomplished, in part, by having each language as similar, as may reasonably be expected, with the next.

Fig. 1 illustrates the ordering of the languages with respect to their descriptive power. First, SPEC* [DASG81b] permits the designer to formally specify the functional aspects of the architecture under design. The architecture is partitioned into integral parts with an input/output criterion being associated with each component. Each component states that for a given input, a specific output will result. Using this language a stepwise refinement is possible which allows the designer to begin with a global view (one component) and gradually increase the information content (many components) of his design.

At some stage in the architectural development the



INCREASING COMPLEXITY
[S*]
SPEC*
FORMAL SPECIFICATION LANGUAGE FOR THE FUNCTIONAL DESCRIPTION OF ARCHITECTURES
S*A
PROCEDURAL LANGUAGE FOR THE PROCEDURAL DESCRIPTION OF ARCHITECTURES
S*
MICROPROGRAMMING LANGUAGE SCHEMA
S*(M1)
MICROPROGRAMMING LANGUAGE IMPLEMENTATION FOR MICRO-ARCHITECTURE OF HOST MACHINE M1

Figure 1. [S*] Family of Design Languages

designer will want to actually indicate how component inputs are transformed to the outputs. This represents an increase in the information content (increasing complexity) from the SPEC* description and is represented by a "program" written in the procedural architecture description language S*A. Taken together SPEC* and S*A represent the design of the architecture.

The implementation of the design is the final and most complex of the three tiers and is based on the high level microprogramming language schema S*. Simply put, this is a partially defined microprogramming language from which the designer constructs a language, S*(M1), for programming the host machine, M1, upon which the architecture is going to be implemented.

The S*A description is then transformed to the implementation language, S*(M1), compiled and run on the host resulting in the realization of the architecture under design.

2.1.2 A Design Approach to An Application Directed Architecture

Applying the concepts introduced in the previous section an informal architectural design methodology for the implementation of a language-directed architecture may now be presented. As Fig. 2 illustrates the design begins with a high level language L1, such as Pascal or C. The instruction set, formats, and support systems such as the calling mechanism and minimum number of registers are carefully laid out. This is a complicated phase involving tradeoffs between such things as code density and opcode optimization [JOHN79]. A functional specification is then constructed for its architectural representation. This is then expanded into a procedural S*A description.

One of two possibilities now presents itself: either a microprogrammable host machine such as the QM-1 will be chosen; or a design automation process will be selected to build the host based on the S*A description. If an existing host is chosen it will, of course, influence the formal specification depending upon its inflexibilities.

In order to microprogram the host a high level language S*(M1) based on S* and the host's capabilities are

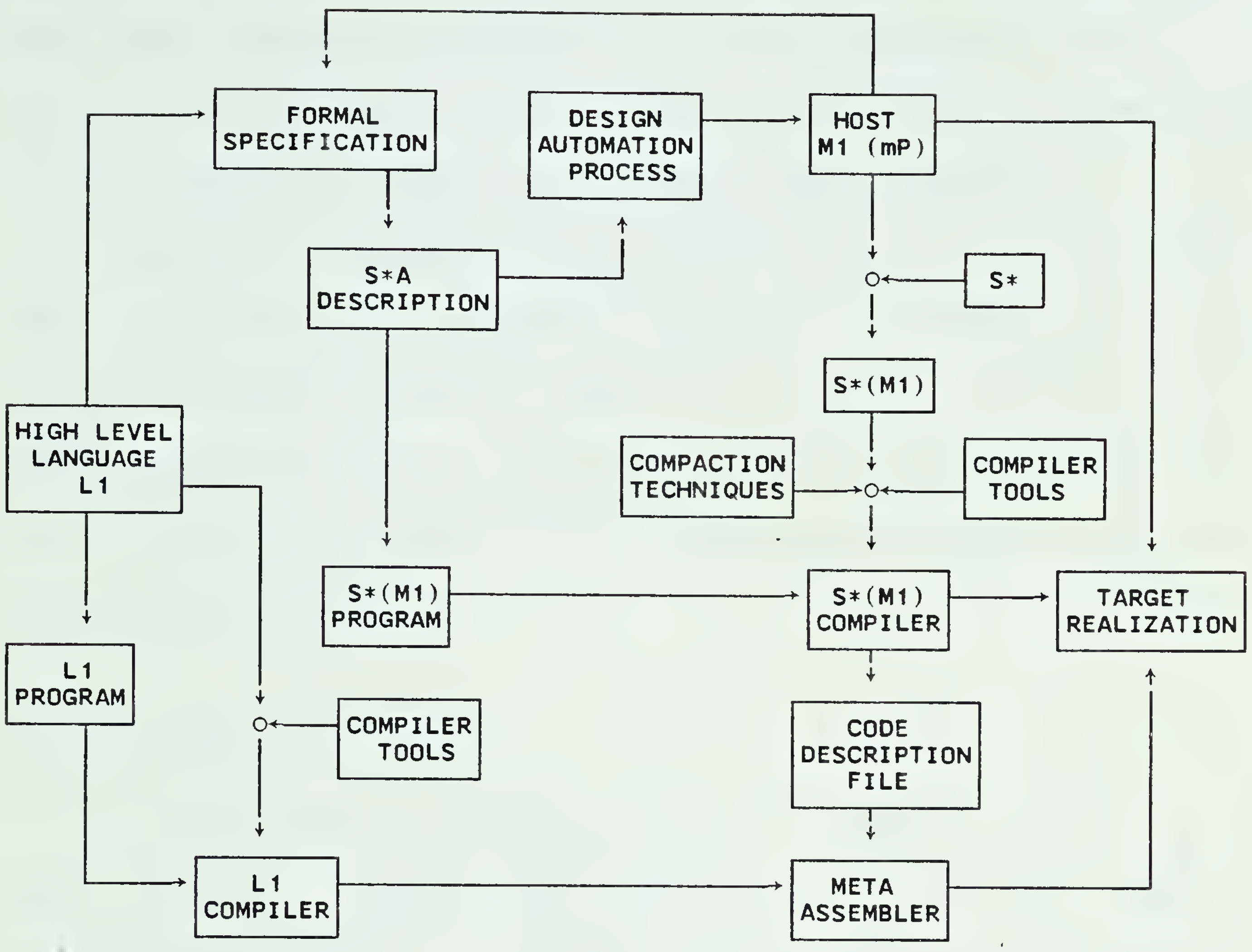


Figure 2. Language Directed Architecture Development

instantiated. A compiler is then developed for S*(M1) using compiler tools, optimization and compaction techniques [LAND80]. The tools will save development time and the compaction techniques are necessary for the production of efficient code without which the compiler would be of little value. We are now in a position to translate, by hand, the S*A description into an S*(M1) implementation program which may be compiled and run on the host thus achieving the realization of the target architecture.

The user of the target machine may now write programs in the language L1 and compile them into an intermediate

language, such as an assembly language. The compiled code is then run through a universal, or META, assembler which uses as one of its inputs a code description file produced by the S*(M1) compiler. This file provides the necessary information: opcodes, mnemonics, instruction sizes, and type of parameters, needed to perform the translation from assembly code to machine code. This file in effect allows the L1 compiler writer and the S*A description implementor to be relatively autonomous in the detailed specification of the instruction set.

2.1.3 Users of S*(M1)

A high level microprogramming language should allow the user freedom to express his algorithms in a clear, easy to understand fashion. This must be balanced with his need to exploit the machine to its fullest extent in the areas of sequential and parallel flow of control and resource usage. The implication of the previous two statements is that the programmer must have the fullest access possible to the machine which leads to the conclusion that the programmer must be as knowledgeable (at least initially) about the machine's characteristics as a microassembler programmer.

We view the S*(M1) representation of algorithms to consist mainly of sequential statements, interwoven with parallel ones. These parallel statements are used only when the operations must be performed in parallel to maintain the correctness of the program. It is then up to the compiler to

compact the resulting (basically sequential) code into the smallest, correctly executable machine code possible. This sequential aspect of programming stems from the author's belief that programmers want to express themselves sequentially and do not want to be overly burdened by the (possibly) parallel aspects of their algorithms.

2.2 The Process of Instantiation

The process of instantiation, illustrated by Fig. 3, is considered to be evolutionary, in that several iterations may be required before an acceptable language is obtained. Four related phases, to be discussed in the following sections, form a method for the design of a language based on S*, the host machine, and a variety of other factors. A method of feedback is established so that experiences gained from a particular instantiation need not be wasted.

Flexibility in approach seems to be a key to success and we have found that the UNIX [KERN81] operating system has allowed us to incorporate methods which permit changes to be made easily and effectively without an undue loss of time.

2.2.1 Goal Establishment

The first priority of an "instantiator" concerns the determination of goals which will act as a guide through the complete instantiation process. They also help to assess, in the final analysis, if the language fulfilled its initial

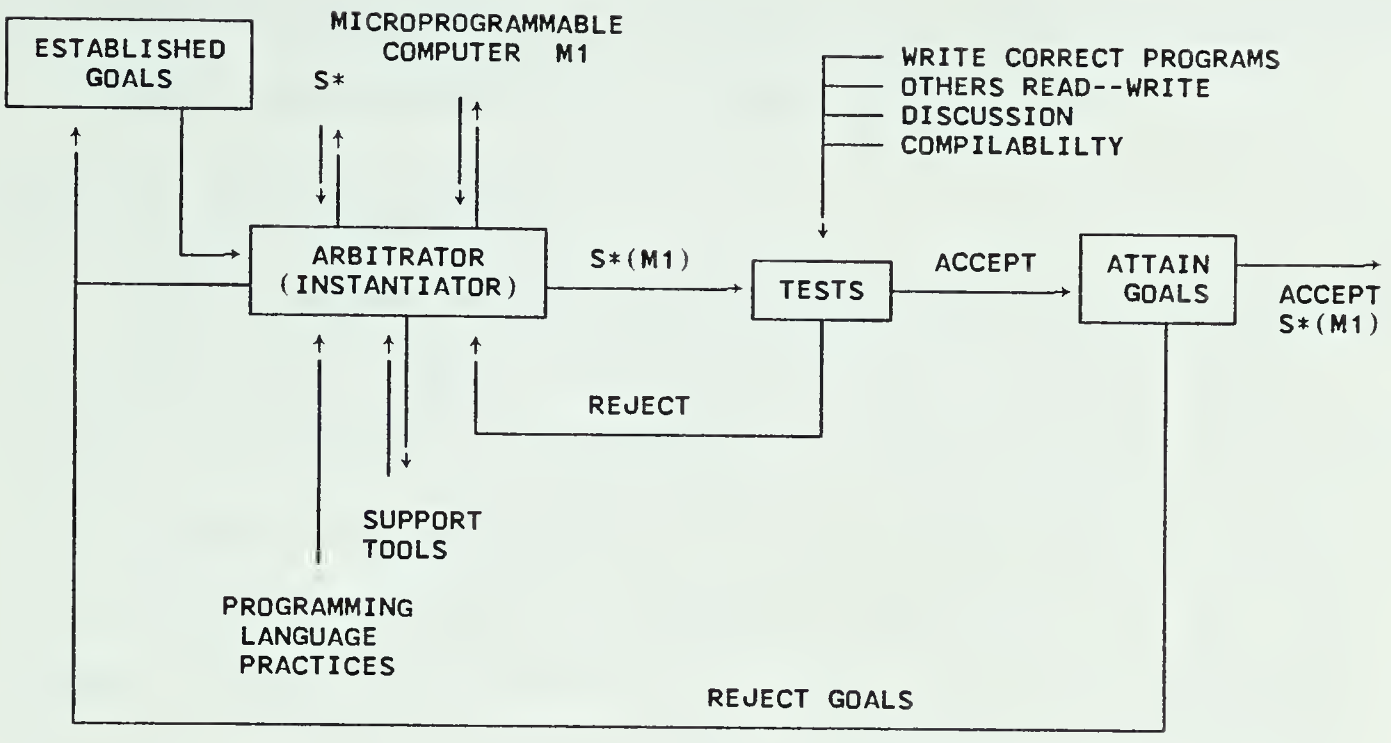


Figure 3. Process of Instantiation

objectives.

The goals established for the S*(QM-1) instantiation are:

1. A programmer using S*(QM-1) must have the capability of writing well structured programs which may exploit the inherent parallelism of the QM-1 at the nano-architectural level, and yet would remain independent of the structure of nanostore and its associated sequencing logic.
2. The programmer, in designing programs, should be able to employ a hierarchical top down approach by which algorithms can be naturally transformed into S*(QM-1).
3. A flexible manner of program representation should be available so that features such as **residual control** can be fully exploited.

4. Finally, programs must be easier to read and understand than programs written in nanoassembler code.

2.2.2 Arbitration Phase

Having established the language design goals we are now in a position to begin the instantiation itself. First, study the constructs and program structuring provided by S*. Then couple this information with the micro-architecture of M1 trying to overlay the concepts supplied by S* onto the capabilities of the machine. M1 should also be considered from the assembly programs that have been written (how it was used), and how it was originally designed to be used.

S* constructs should now be mapped onto M1 with machine dependent constructs and statements added to complete the language specification. We take the view that initially the compiler should not retain any decision making duties, i.e. which ALU to use or which data path to select. Once experience has been gained a decision may be made relax this restriction allowing the compiler to perform certain decision activities, although with the understanding that no limitations should be imposed on the programmer.

Support tools now come into play, the advantages of which must be closely weighed against any restrictions they may impose. For example, in the instantiation of S*(QM-1) the UNIX tools LEX [LESK79] and YACC [JOHN80] have been selected to aid in the construction of a parser and lexical analyser. The only restriction imposed was S*(QM-1) had to be in the class of LR(1) grammars [AHO79], which was not

considered to be a major drawback.

The compaction and optimization techniques which are going to be employed in the compiler should also be considered, since the language may not allow enough programmer control, i.e. it is too general, for effective compaction or optimization to be performed.

2.2.3 The Testing Phase

The testing phase is important as the instantiator may, by this point in time, be locked into a mental set and feel that the language is quite acceptable when in fact it is not. The first step is to write correct programs and check them syntactically against the definition of the language. The compiler tools allowed the grammar to be built up in a manner (similar to a BNF definition) which could be used as an input specification for YACC and LEX so that the generated parser could perform syntactic analysis, at the same time as the language was being developed. The parser was modified to give better diagnostics of its actions, allowing errors in design and implementation to be quickly identified and changed.

A decision can now be made to accept or reject the instantiation. If the language does not meet the original design goals then either the goals are too broad or the language is not acceptable. It may also turn out that the goals were not broad enough and should be expanded to meet the instantiated language.

2.2.4 Feedback Phase

After having gained experience using the language schema, support tools and microprogrammable machine we feel it is the user's responsibility to provide feedback to the originators. This involves such things as additional constructs for S* resulting from new architectures or technological changes, or suggested changes to the microprogrammable computer which the original manufacturer did not envisage.

2.3 S* Constructs and Some Recommended Changes

The objective of this section is to review S* as it has been presented in three papers by Dasgupta (see references) entitled: "Towards a Microprogramming Language Schema"(1978), "Some Implications of Programming Methodology for Microprogramming Language Design"(1980), and "Some Aspects of High Level Microprogramming"(1980). Based on this review we will conclude with some recommended changes to S* which have have been incorporated into S*(QM-1), the syntax and semantics of which may be found in Appendix 1. In order to be concise and accurate and to aid in clarity, quotes in this section will be referred to directly, based solely on the page number provided.

S* has been presented to the public in a semi-complete format with many formal syntactic and semantic definitions. On one hand this is a sensible approach as it allows some experience to be gained before S* is formally 'cast in

stone'. On the other hand, without a formal syntactic and semantic document (schema definition) the potential instantiator is left with some discrepancies due to the construct and definition changes between the earlier paper (1978) and the later ones (1980). We shall discuss these points in detail and attempt to present a balanced and clear view of S*.

The approach taken is top-down, beginning with a program definition, the basic form of which is **prog** {name}; <declaration-block>; (also known as <decln-block>) <execution-block> **end**.

2.3.1 The Declaration Block

The declaration block is defined variously as "a sequence of one or more data-object/synonym/procedure declarations"(151), or "is a sequence of declarations"(317). This block functions as a means of describing the resources of the machine to which the programmer will be referring within his program, that is "a microprogrammer must declare all data-objects before referencing them"(313). A resource is referred to as a variable data-object and is defined in terms of a data type. Data types, which are built up from the primitive data type--a bit (0,1), may be structured in terms of sequences of bits, arrays and associative arrays of more primitive types; stacks which permit only the top element of the resource to be accessed, and finally tuples(as in PASCAL records) which are resources consisting

of two or more variables of possibly different types grouped together under one name. An optional pointer, **with** ..., may be used to specify which data objects are used to access the stack or array. Also, a special class of variables called synchronizers is employed for synchronizing microprocesses.

Pseudovariables provide an abstraction of variable data objects which allow non-variable resources, such as bits in a microinstruction control field, to be used in the same manner as variables.

Once declared a variable data-object may be renamed by using the synonym declaration which may only appear in the declaration section or in a <proc-head> (to be explained below). Constant data objects are pre-defined, invariant and "correspond to a machine defined literal constant located in a read-only memory element"(146).

A procedure has the form: **proc** name (<par>{,<par>}); {<proc-head>}; <proc-body> **retn**. The description provided is: "Data-object names referenced within <proc-body> must either be parameter names or their locally defined synonyms"(150) or alternatively "all data-objects referenced in <proc-body> are global objects"(317), and also "<proc-body> is either a sequential or region microstatement"(150).

2.3.2 The Execution Block

The execution block consists of "a sequence of one or more regions, sequential microstatements or program

blocks"(151), or "is a sequence of executable statements"(317). Executable statements are divided into two groups: single microstatements, in turn divided into five subtypes which are dependent to some degree upon the instantiation; and composite statements consisting of parallel microstatements, or one of four other groupings of statements which are all fixed in S*.

Simple statements are listed variously in "three classes"(146), or "four types"(314), and consist of transfer, function, primitive (simple) selection, procedure call and goto statements. The transfer statement is concerned with the direct data transfer from one resource to one or more other resources, whereas the function statement deals with data transformations determined by "primitive machine-defined operations"(146). We are informed that "primitive operators in the instantiated language are mostly contained in the operator set defined in S*"(314) however these operators are not given in the literature.

The primitive selection is an order-independent, multiple **if** statement using single clauses, not unlike the PASCAL case statement, in which at most one of the multiple if conditions may be true at any one time. The action associated with a true test is a simple statement which may not be a primitive selection statement.

The method of procedure call varies from the earlier to the later papers. Initially we are informed that a procedure is invoked (called) by the "statement **call** "name" where

"name" is a procedure name"(150) and later "A procedure call is simply denoted by a ... procedure name with or without parameters"(315) which indicates that the reserved word **call** is no longer required.

Parallel constructs begin as the simple concurrent microstatement **cobegin** ... **coend**, the local sequential parallel microstatement **cocycle** ... **end**, and finally the extended concurrent microstatement **dur** ... **do** ... **end**. These were deleted, modified and expanded (without the reader being explicitly informed) until there now exists a **cocycle** ... **coend** and **stcycle** ... **end** statement, where the cocycle takes the place of the old **cobegin** and **cocycle**, and **stcycle** replaces the **dur** construct. The **cocycle** handles the cases, of parallel and sequential micro-operations within a single microcycle and **stcycle** handles extended concurrency.

The use of '□' (a new delimiter introduced in the 1980 paper) and ';' within a **cocycle** or **stcycle** indicates in the former that for a **cocycle** the statements must be executed together within a single microcycle and for the **stcycle** are to begin in the same microcycle but the time to termination is left unspecified. The ';' indicates for the **cocycle** that the statements are to be executed in a strict sequential ordering within a single microcycle, and for the **stcycle** the statements must begin in the specified order within a single microcycle but their termination is left unspecified.

The sequential microstatement **begin** ... **end** and **region** ... **end** constructs are basically directives to the compiler

that in the first case, statements within the construct should be optimized and in the second case the statements are order dependent and the sequential flow of control is to remain unchanged.

The generalized selection statement "which is simply a generalization of the primitive selection statement"(316) is introduced but not elaborated upon. The same holds true for the "parentheses"(316) **do** ... **od**. Finally we have the standard repetition statements **while** ... **do** ... and **repeat** ... **until** ...

2.3.3 Recommended Changes to S*

The following catalog of recommended changes to S* is based on the comments of the previous section and the experience gained with S* in the instantiation of S*(QM-1).

2.3.3.1 Reserved Words

We recommend that the **end** reserved word be expanded wherever possible to aid in the readability of a program, for example a program should be terminated by **endprog**, **stcycle** by **stend**, **while** by **endwhile**. A complete syntactic list of reserved words used in S*(QM-1) may be found in Appendix 1.

2.3.3.2 Pre-defined Variables

All types, variable data-objects and machine-dependent constants should be designated by the instantiator (or

compiler writer) when S* is implemented for a particular machine. They would thus constitute a formal pre-defined description of the machine indicating the totality of resources to which a programmer has access. In order to aid in the description of resources and do away with "host-machine defined"(146) synonyms an expanded method of data-object declaration is proposed where a single resource may be declared as a multiple grouping of type structures. This method of description is used in the S*(QM-1) pre-defined data-objects to be found in Appendix 2.

To provide the reader with some idea of the nature of declarations, and to indicate the new method of specifying differently "perceived" register variables the structure of a register bank called *local store* is presented:

```
type ls_register = seq [ 17..0 ] bit
type f_register  = seq [ 5..0 ] bit

var local_store

    : array [ 0..31 ] of ls_register
          with fmod, fcod, faod, fsod, feod, gspec

    : tuple
       general_purpose  : array [ 0..23 ] of ls_register

       index          : array [ 0..3 ] of ls_register
                        with fmpc

       general_purpose2 : array [ 0..2 ] of ls_register

       instruction_reg  :  ls_register

                        :  tuple
                            c : f_register
                            a : f_register
                            b : f_register
                           endtup

                        :  tuple
```

```
                              opcode      : seq [ 6..0 ] bit
                              a_parameter : seq [ 4..0 ] bit
                              b_parameter : seq [ 5..0 ] bit
                             endtup
    endtup
```

The method of referencing this type of multiple definition is as expected. We can say **local_store[2]** to directly access the array data type or **local_store.index[3]** or **local_store.instruction_reg.b** to access a tuple member. Where there is no naming conflict the tuple header name may be dropped as in **instruction_reg.b** or simply **b**. The **with** pointer is used as an index for entering data into an array. **Local_store[feod]** assumes that F-register FEOD (to be discussed later) has been assigned a value between zero and thirty-one; note that it is incorrect to say **local_store.general_purpose[faod]** as this pointer is not associated with that particular portion of the variable local_store.

The scope of these pre-defined data-objects is global and the programmer retains the ability to rename them as he chooses using the synonym declaration, to aid in textual clarity. These synonym names are global if declared in the declaration block and local to a procedure if declared in a procedure head.

The programmer should also be able to define literal constants in the same manner that invariant read-only memory constants are declared by the instantiator.

A new construct should be added, **declaration ... endec**, to delineate the declaration block.

2.3.3.3 Procedures

A new reserved word **endproc** should be used in place of **retn** which in turn should be expanded to **return** in order to signify a return is to be made from the called procedure. With the incorporation of **declaration** ... **endec** construct we recommend that procedure declarations should be made after **endec** but before any executable statements. We feel that the **begin** ... **end** construct is superfluous within the bounds of a procedure and may be left out if desired. The method of invoking a procedure is clarified in the next section.

2.3.3.4 Branch Statements

A new type of executable statement should be incorporated into S* consisting of the reserved words: **call**, **return**, **act** and **goto**. The **call** statement (**call** "name" where name is the name of a procedure) is used to call a procedure in such a way that control is returned to the next statement on procedure termination either through "falling out the bottom" (**endproc**) or after an explicit **return** statement is encountered.

act is used in the same manner as **call** and serves to activate a procedure with the understanding that control will not be returned, and as such it is up to the procedure to carry on the flow of control (this is essentially a **goto** to a procedure). The **goto** may only be used to transfer control within a procedure or execution block.

2.3.3.5 Statement Sequencing

The last recommendation concerns statement sequencing in relationship to the role of a compactor. We feel that the statement separator ';' should be limited solely to executable statements where its use will have a more pronounced significance. Its normal function will be as a weak sequential delineator indicating to the compactor that the statements are to be compacted as well as timing, data dependencies and resource conflicts will allow. Its use within the region construct indicating a strict sequential flow of control is to be maintained within the construct since the programmer may know of some conflict which the compactor would not be able to discover due to, possibly, *residual control* effects.

Once several statements are bound together by a **region**, **stcycle**, or **cocycle** constructs they may be *compacted up* into previous statements, or have statements compacted into them with the provision that the timing specifications within the construct are not be disturbed, and no other conflicts arise. As an example consider the following partial program:

```
S1 ;  S2

cocycle
     S3 □ S4 □ S5
coend ;

S6 ;

region
     S7 ; S8 ;
     cocycle
          S9 □ S10
     coend ;
     S11
```

```
        endreg ;
        S12 ;
```

where the S's represent single statements. Statements S3, S4 and S5 must execute within the same microcycle and if there are no conflicts (data dependencies, resource contentions, or timing constraints) with S2, they may all be executed together (compacted up). S7 must complete execution before S8 commences and S9, S10 must execute together and may not start until S8 finishes. If no conflicts arise S7 may be executed at the same time as S6, and similarly S12 may be executed with S11 or even S7 if no conflicts arise with any intervening statements.

Chapter 3
An Architectural Investigation of the QM-1

This chapter presents the reader with a model of the QM-1 at the nano-architectural, or component and data-path level as seen by a nanoprogrammer. The intention is to provide a global view of the QM-1 in order that the reasoning behind certain S*(QM-1) design decisions may be understood. Additional background material may be found in [AGRA76, SALI76]. The QM-1 Hardware User's Manual [NANO79] should be consulted for a more detailed account of the QM-1.

3.1 Overview

The QM-1 was designed as a user microprogrammable computer for **indirect emulation** of *environments* and existing computers and for the implementation of higher level languages, among other things [ROSI72]. As Fig. 4 demonstrates it is centered around *local store*, thirty-two 18-bit registers; and *F-store*, thirty-two 6-bit registers, which provide **residual control** and 6-bit temporary storage. The 32nd local store register, R31, also acts as three 6-bit, (C, A and B) registers for communication with F-store.

The designers felt that, when emulating a computer, setup or residual control registers could be used to control data transfer paths, acting solely as indexes into local or

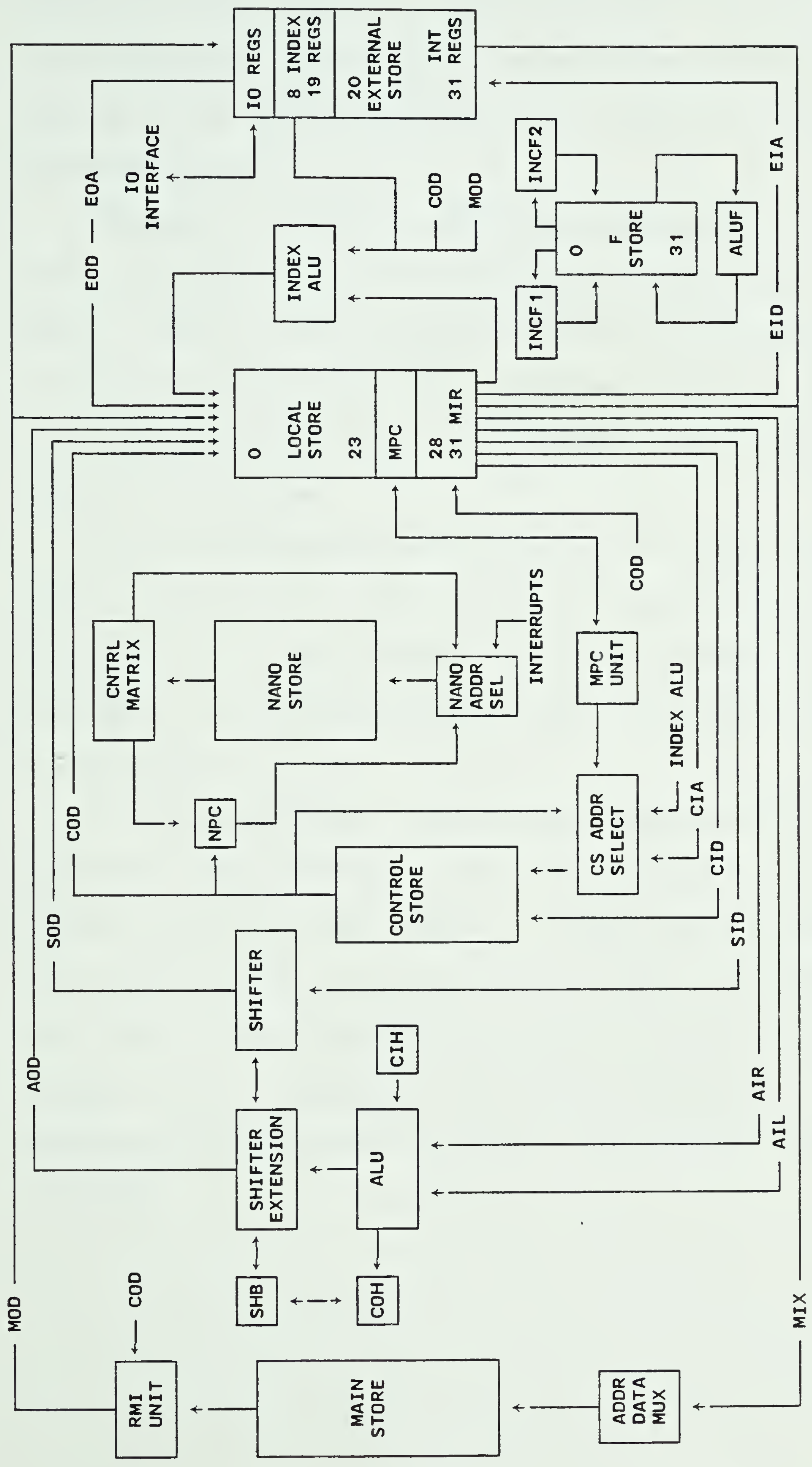


Figure 4. QM-1 Functional Components and Data Paths

external store. Once set, these registers would remain fairly invariant throughout the emulation, thus negating the need to place source and destination register fields in the nanoword [FLYN71].

The data transformers include a 6-bit ALU, two 18-bit ALUs, a shifter capable of eighteen or 36-bit shifts and a special local store incrementing facility. *External store*, thirty-two 18-bit registers, provides interrupt addressing and enabling, 18-bit constant storage, and an eight register external interface.

18-bit data buses are indicated, in Fig. 4, by lines joining components, with the data flow direction shown by an arrow. Each of these components operate in parallel and may send data to, or receive data from, local store independent of any other.

The QM-1 employs a three level memory structure in which the target machine, with its instructions in main store (18-bit 800 nsec memory), is defined and interpreted by a microprogram in control store (18-bit 164 nsec memory). Each 18-bit microinstruction, consisting of a fixed 7 bit opcode, a 5-bit parameter and a 6-bit parameter, is in turn defined and interpreted by one or more 360-bit nanowords [NANO79].

3.2 Nanostore and Control Store

Nanostore contains up to 1024 360-bit nanowords, arranged in segments of 128 word pages. Each word consists of five 72-bit partitions: a K-vector and four T-vectors, T1 through T4. A priority selection mechanism determines which nanoword is to be selected for gating into the control matrix, where it becomes the active nanoword.

When a nanoword becomes active eight 6-bit constant fields from the K-vector, which may be considered to be stored in registers, are activated providing two constants, ALU and shifter control signals, and three test masks. Alternatively, the control signal and mask registers may be used to house constants if their primary function is not required. The remaining twenty-four bits of the K-vector are used for additional control such as for conditional branch addressing (10 bits), and enabling interrupts.

The constant "registers" retain their contents during the time a nanoword is active and are immediately replaced with the values in the next nanoword when it is gated into the control matrix. These 6-bit constants are the only means a nanoprogrammer has of introducing constants into registers directly from nanostore.

At any time only a single 72-bit T-vector is active and as such its control fields (nanoprimitives), many of which are further decoded by the control matrix, provide the logic signals required to control the machine. A T-vector is active for 82 nsec (this time span may be doubled if the

stretch bit is set in the T-vector) and on completion the next T-vector in the sequence T1-T2-T3-T4-T1...., becomes active. This cyclic process continues until a new nanoword is gated into the control matrix. A single T-vector may be conditionally bypassed using the nanoprimitive *skip*, as explained below. The skipped T-vector is still *activated*, that is 82 nsec elapse before the next T-vector, but its actions are inhibited.

On the nanoprimitive signal *read ns* the priority mechanism determines the next nanoword to be read, and places it on the nanostore output lines. The highest priority address is the 10-bit *next address field* in the active K-vector and it is chosen if the *branch bit* in the K-vector is set. The next alternative consists of thirty interrupts, arranged in descending order, the addresses of which are held in a condensed form (6-bits) in external store registers (22-31). To be selected the interrupt must be latched, enabled--by previously having set a bit in an appropriate external store register (18-19), and allowed by having a bit set in the active K-vector. As a last resort the nanoprogram counter, NPC, will be selected.

Once the selected nanoword is ready it may be conditionally or unconditionally gated into the control matrix. If a conditional gate is requested the specified mask, a K-vector control field, is ANDed with the appropriate test bits and a true result causes the gate to occur making the nanoword active. If the result is false the

nanoword is not gated and the next cyclic T-vector becomes active. A conditional branch is obtained by setting the *branch bit* and *alt branch bit*, which cycles the *branch bit* each time a *read ns* is encountered. Thus a false conditional gate test (no gate) followed by a *read ns* causes the NPC to be selected as opposed to the *next address field*.

The NPC may be loaded in one of three ways: from the *next address field* in the active K-vector, sequentially by adding one to itself, and from the control store output data bus (COD). Specifying the COD causes three page pointer bits from a special F-store register, FIDX, to be concatenated with the seven high order bits on the COD and placed into the NPC. The low-order eleven bits are saved in a dedicated register for later loading into the instruction register, local store register thirty-one, under programmer control using the nanoprimitive *Load R31*.

This method of loading the NPC provides an 18-bit parameterized shorthand for invoking one or more nanowords which will interpret the instruction. Therefore, it is seen that microinstructions in control store are interpreted by one or more 360-bit nanowords in nanostore, and that the maximum number of different instructions obtainable, without extraneous escapes by changing FIDX, is 128.

Extending this concept one step further, it is easily seen that mainstore instructions may be decoded and executed by one or more control store routines. Limitations do not exist concerning instruction format or size. This scheme is

known as **indirect emulation** as opposed to **direct emulation** which uses nanoprograms to directly execute mainstore instructions. The advantage of indirect emulation is that the programming task is broken into two more easily managed divisions at the cost of a (possibly) slower emulation.

A nanoprimitive, *write ns*, is available for writing data into nanostore. The B field of R31 specifies which of twenty 18-bit bytes in a nanoword will receive the data which is taken from the EOD bus. The nanoword address is formed in the low order ten bits of the CA fields of R31. The modified nanoword is at the same time read from nanostore in a manner similar to the *read ns* nanoprimitive. If an illegal byte (>19) is specified, the write operation does not change anything and the addressed nanoword is read from nanostore in readiness for gating into the control matrix.

3.3 F-Store and Residual Control

The first fourteen F-store registers provide **residual control** over the 18-bit bussing structure of the QM-1. The contents of an F-register indicate which local store, and in some cases external store, register is to be connected to the bus controlled by it. If the 18-bit register is a *source*, i.e. an input data/address bus "ID, IA", its contents are available to the bus and remain so until the contents of the F-register is changed.

If the 18-bit register acts as a *sink*, i.e. from an output data bus "OD", the data on the bus must be explicitly gated into the register with the appropriate (one of six) *gate* nanoprimitive. This gating procedure allows all data transformers to be asynchronous, that is always in operation, using data presently available on their input bus(es) placing the results onto the output bus. Data operations are controlled by the use of programmed timing delays which allow time for the correct results to propagate through the transformer to the output bus *before* it is gated into a local store register.

There is a discrepancy between the number of local store registers, thirty-two, and range available to an F-register, sixty-four. The following data values result when a value greater than thirty-one is placed in the associated F-register: the buses CID, CIA, MIX, EID, AIR, AIL and SID each take values of all ones and the EOD bus is set to zero. The F-register FMOD, controlling the mainstore output bus MOD, is used in such a manner that the values 0-31 refer to the local store registers and the values 32-39 refer to the first eight external store registers. A value greater than thirty-nine results in a null operation, that is, the data is transferred to nowhere. The remaining four buses are used solely as inputs to local store registers and as such ignore the high order bit in the controlling F-register.

The six F-registers, 14-19, provide additional control

signals such as the nanostore page pointer, the local store register (23-27) to be used as the microprogram counter, global status conditions and a special *phantom* register. The remaining eleven registers are known as the *g-registers* and provide 6-bit temporary storage. A 6-bit ALU provides F-store with sixteen standard arithmetic operations and in addition two single register incrementers (f+1, f-1) are available.

Values for these registers may only be introduced via constants in the active K-vector or through the three 6-bit fields in R31.

3.4 Local Store and Data Transformations

For the purpose of data operations local store may be conveniently represented as a single array of thirty-two registers for use with the ALU and shifter. In addition it may be viewed as twenty-eight registers, which act as sinks and sources for the index ALU, surrounding four register (23-24) which are connected to the microprogram counter incrementing unit.

3.4.1 Alu and Shifter

The main ALU, as shown in Fig. 4, has two input buses AIR and AIL (alu input right/left) and is continuously performing one of thirty-two (16 logical and 16 arithmetic) operations determined by the value in the K-vector "register" KALC. Four status bits: carry, sign, result and

overflow are continuously being produced and may be tested for conditional *skipping* or *gating* with the K-vector mask KT. The local test bits may be transferred, under program control, to the global test F-store register, FIST, for later testing, if the *alu status enable* bit is set in the active K-vector. FIST may be tested using the K-vector test mask KS.

The shifter has one input bus, SID (shifter input data), and is controlled by two K-vector "registers", KSHC (shifter control) and KSHA (shift amount). The shifter operations consist of left and right--circular, logical and arithmetic shifts of either eighteen or 36-bits. If a double shift (36-bit) is selected the shifter extension with data from the ALU is used for the high-order 18-bits. The amount of the shift is specified in KSHA with the correct modulo value being used to suit the type of control. A shift of zero bits simply passes the data directly through the shifter.

Two local condition bits, SLB and SHB (the low/high bits present on the SOD bus) are available for testing with the mask KT. These bits may also be transferred to FIST if the *shifter status enable* bit is set in the active K-vector.

A 16-bit mode is available for both the shifter and ALU by setting the high-order bit in the special F-register, FIDX. This may be overridden with a bit in the active K-vector, *force 18-bit mode*. In the case of the ALU the two high-order bits on the input buses are automatically

replicated from their respective third high order bits, the sign of the 16-bit value. The shifter performs operations on the low order sixteen or thirty-two bits of data. Also, the local conditions: sign, result, overflow and SHB are redefined to reflect the 16-bit mode.

Two flip-flops, CIH (carry in hold) and COH (carry out hold) are used for carry control with the ALU and shifter apparatus, and are controlled by a 3-bit field in the active T-vector. Each may be set or cleared directly. The CIH is always used as a carry in to the ALU and may also be set with the carry value last generated by the alu. The COH may be loaded with the latest carry value (it becomes the local carry test), or loaded with the low order bit of the SID bus, or from the *shifter end bit* which is the last bit shifted out of the high end of the shifter extension.

3.4.2 MPC Unit

Fig. 5 provides a detailed model of the MPC unit and the associated control store address mechanism. The component is asynchronous, continuously placing on its four output lines the local store register, as selected by the F-store register, FMPC (mod 4), plus: one, two, the 6-bit B field in R31 and the low order eleven bits of R31.

The single input and four outputs may be used directly as control store addresses, as determined by the *control store address select* field in the active T-vector, along with the data on the index ALU output data bus, the contents

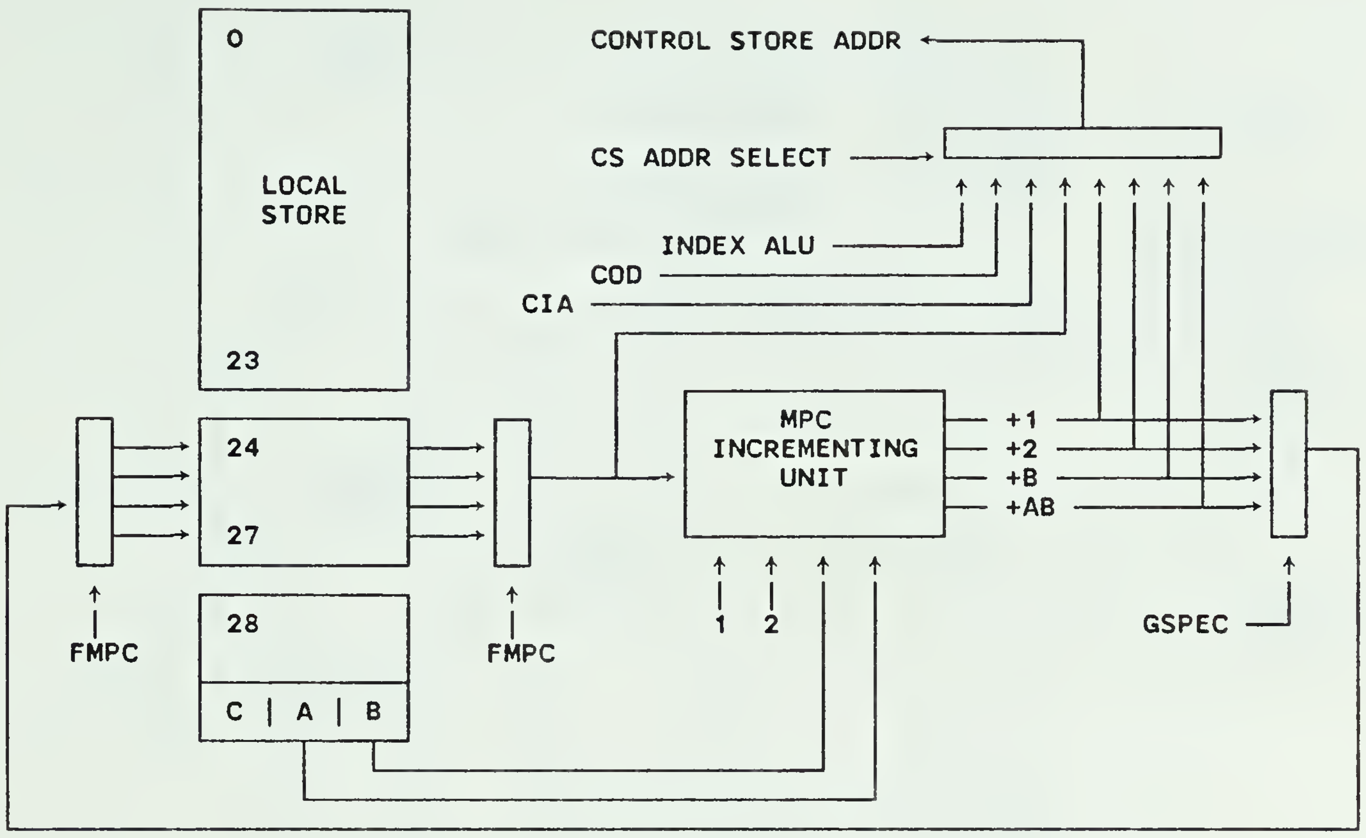


Figure 5. MPC Incrementing Unit

of the control store output data bus and finally the contents of the local store register indexed by FCIA.

One of the four outputs is selected by the low order two bits from a 4-bit field, *gspec*, in the active T-vector. The result will be gated into the local store register indexed by FMPC if the *inc mpc gate* bit is also set.

3.4.3 Index Alu

The index ALU, Fig. 6, is the most complicated data manipulator in the QM-1 as it uses one, two and sometimes three levels of indirection to determine the inputs, operation and output data destination. Its logical and arithmetic operations are computationally equivalent to the ALU. Use of the A and B parameter fields in R31 makes this

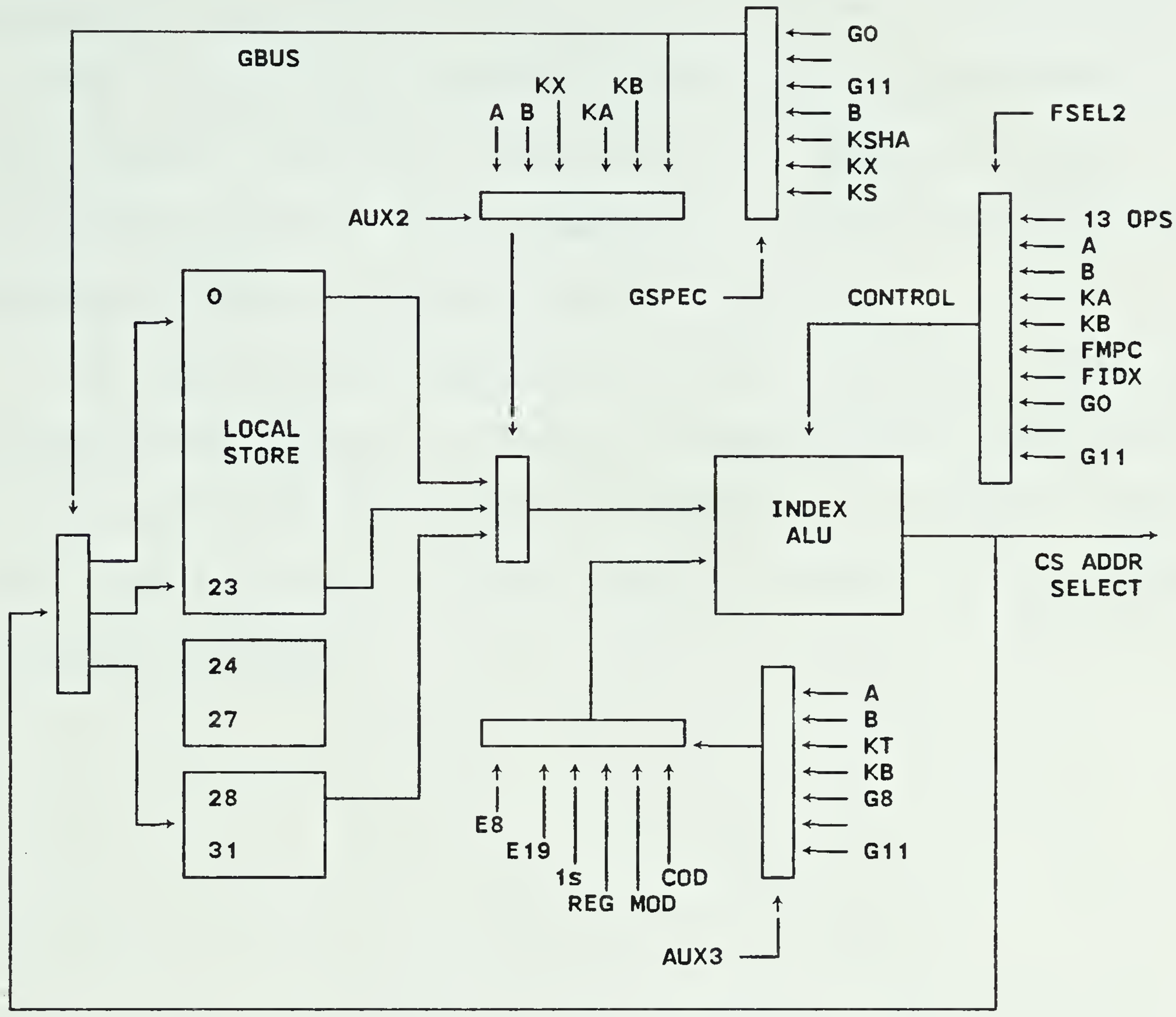


Figure 6. Operation of the Index Alu

unit particularly attractive for quick index operations required by control store instructions.

Four fields in the active T-vector control the operation of the index ALU. The field *aux2* selects one of six 6-bit inputs to a multiplexor which determines the local store register selected as the left input. A special case is the *gbus* which requires a further level of indirection through the 4-bit *gspec* field. The right input is one of: twelve external store registers (8-19), a special CPU control register, a source of all ones and the data on either the mainstore or control store output buses; and is

selected by the 6-bit contents of a register designated by the *aux3* field. The arithmetic operation to be performed is controlled by the *fsel2* field which may select thirteen operations directly, or 32 operations indirectly through the contents of one of eighteen 6-bit registers.

The local store register to receive the ALU result data is specified by the six bits on the gbus and will be gated if the *index gate* bit is set in the active T-vector. Recall, also, that the result may be selected for use as an address into control store.

Chapter 4

An Instantiation of S* to S*(QM-1)

4.1 Introduction

In keeping with the S* objective of constructing and understanding microprograms which are independent of control store organization [DASG78] (nanoprograms in nanostore) and the fact that the QM-1 was designed for indirect emulations (section 3.1) we have instantiated a language, S*(QM-1), which is solely procedure based. As mentioned in the previous chapter nanostore acts as a control store interpreter. It does so using only three types of *nanoprogram routines*: control store instruction interpreters, interrupt handlers, and subroutines which provide support for the interpreters and handlers. For each of these three nanoprogram types we have proposed a different procedure which informs the compiler where to position the nanowords within nanostore, the method of transferring control from one nanoword to the next within the procedure, and what, if any, auxiliary actions need to be performed.

As this is the first version of S*(QM-1) we have developed a language in which the programmer exercises absolute program control over the QM-1 leaving the compiler with no decision making capabilities.

Fig. 7 illustrates the general structure of S*(QM-1).

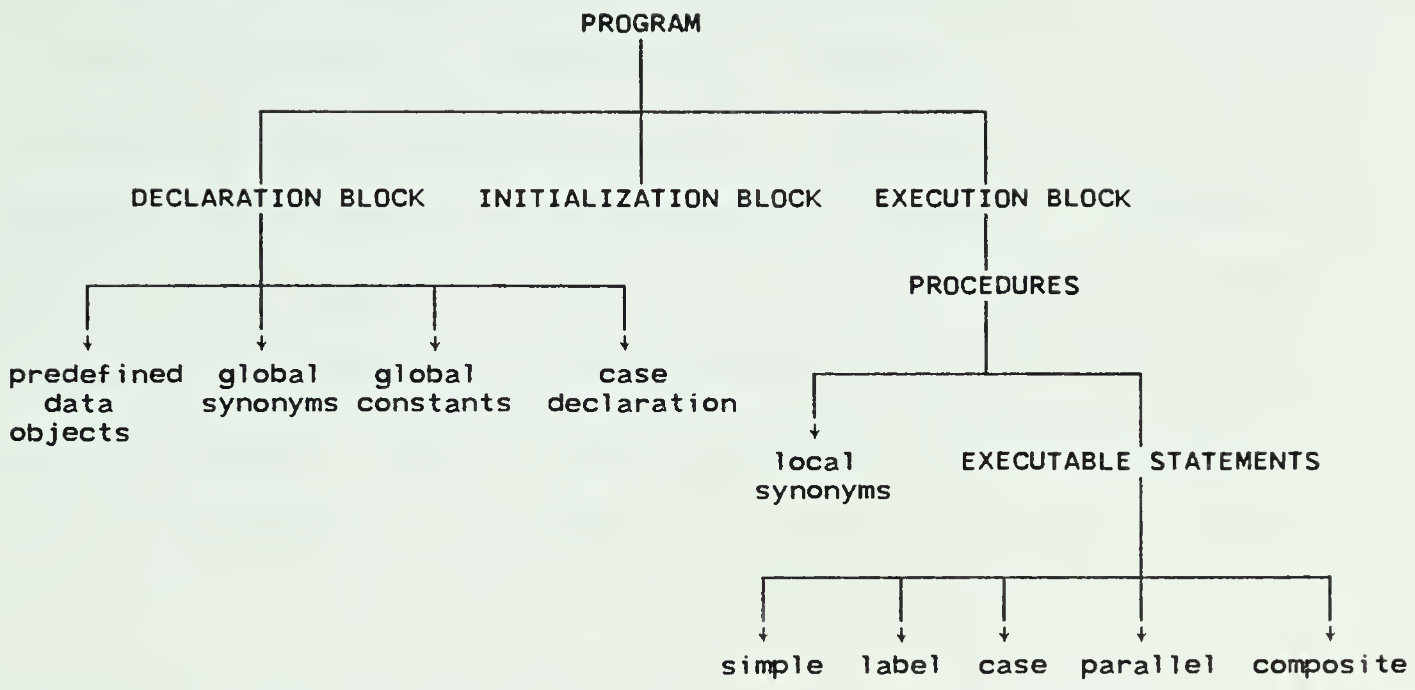


Figure 7. S*(QM-1) Language Model

```
prog ( ... )
          declaration
             .
             .
             .
          endec
          init
             .
             .
             .
          endinit
                            ┌ instruction, ...
          proc ... (  ──────┤ interrupt, ...      )
             .              └ subroutine, ...
             .
             .
          endproc
            ┌
            │  additional procedures
            └
endprog
```

Figure 8. A Skeleton Program

Fig. 8 demonstrates the basic organization of a program written in S*(QM-1). Each of the "blocks" will be explored in the following three sections. A complete syntactic and semantic description is included in Appendix 1. Appendix 3 provides the reader with a short program example. A more extensive example based on the mode calculations for a PDP-11 emulator [DEMC76] may be found in the technical report on S*(QM-1) [KLAS81].

4.2 The Declaration Block

As Figs. 7 and 9 reveal, the declaration block consists of four divisions which may in turn be divided into two groups: pre-defined data objects; and synonym, constant and case declarations. The case declaration is a new construct and will be described in a later section.

Pre-defined data objects consist of variables, **var**'s and pseudo variables, **pvar**'s, to which the programmer has access. They are global in scope, and are defined in terms of bits, sequences of bits, types, arrays and tuples. Appendix 2 provides a full listing of these data objects.

The number of psuedo variables has been limited to the fewest number possible. They are not true variables in the sense that they may hold a range of values or may have their values transferred to another variable, but are in fact abstractions of certain fields within a nanoword. In the case of **pvar**'s the type **bit** indicates the **pvar** may be set to one or zero, and a sequence indicates that a range of values

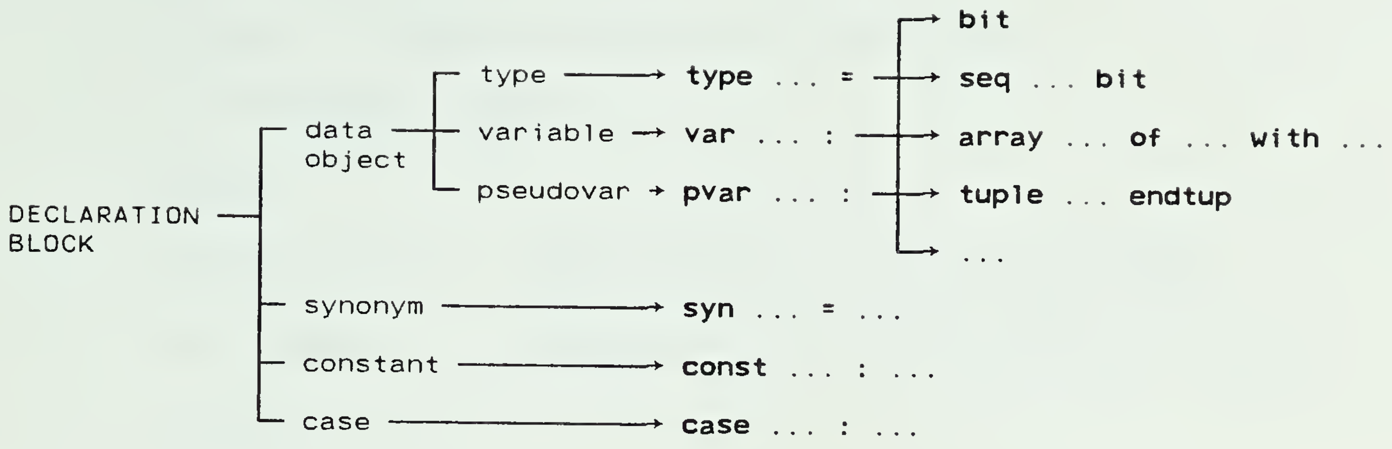


Figure 9. Declaration Block

may be used. These **pvar**'s must always be on the left hand side (*sink*) of an assignment (transfer) statement.

A perusal of Appendix 2 shows that a variable identifier such as *local_store* may be used in several different contexts depending on the bus in question. For example:

external_store_input_data.local_store[5]

refers to the same register as

control_store_address_source.register_address.local_store[5]

but in the former case the bus connects local store to external store via the EIA bus and in the latter to the control store address bus, CIA. As discussed in section 3.3 residual control registers may contain values exceeding the number of local store registers in which case either the bus is set to a value of *all zeros*, or *all ones*. This is made clear by the tuple construct enclosing local store followed by all_ones, or all_zeros, which effectively describes the array directly preceding it. For example:

```
var alu_input
   : array [ 0..63 ] of seq [ 17..0 ] bit
      with fair, fail
   : tuple
      local_store : array [ 0..31 ] of ls_register
      all_ones    : array [ 0..31 ] of source_all_ones
     endtup
```

Synonyms in the declaration block allow programmers to globally rename pre-defined data objects so that more appropriate variable identifiers may be employed. Appendix 3 provides several examples of how the construct is used. A range of registers may be renamed as in:

```
syn pdp11_register = local_store[0..7]
```

which indicates that the first eight local store registers may be referenced by the variable identifier *pdp11_register*. The programmer is responsible for his own bounds checking, as it is unreasonable, due to space restrictions, for the compiler to include run time checking. Due to the effects of residual control registers the lower bound must always be zero.

Constants and literals are the only means a programmer has of introducing values into registers from an S*(QM-1) program. The reason for this, as explained in sections 3.2 and 3.3, is that they must be encoded (by the compiler) into a K-vector constant field. Thus they may not exceed 2^6-1 (63), which implies that they may only be assigned to 6-bit

registers, that is, of **type** *f_register* or *k_vector_register*. The obvious exception is when literals or constants are assigned to **pvar**'s which have their own restrictions.

Constants have a bracketed decimal number associated with them to indicate the number of bits the value is to be encoded into, always six for **var**'s and either one or two for **pvar**'s. Constant declarations and literals have a modifier: **bin**, **oct**, **dec** or **hex** to indicate the radix of the number. If the modifier for a literal is not present a decimal radix is assumed.

Examples:

```
const left_and_not_right  : bin (6) 10
fail := 30;  fail := bin 1101;
local_store[10] := 7654;  /* incorrect */
fair := 72;               /* incorrect */
```

4.3 Initialization Block

When using S*(QM-1) the programmer requires a method for initializing registers before emulation begins. For example the local store register to be used as the microprogram counter must be established, bus control F-registers may need to be initialized, and external store constants for use with the index ALU may need to be setup. To provide the programmer with this facility we have incorporated a new construct into S*(QM-1):

```
init    ...    endinit
```

which follows the declaration block.

The register initializations and interrupt procedure specifications, which are used to setup interrupt bits and address fields in external store, are placed in a table by the compiler. The last action of the QM-1 loader, immediately before execution begins, is to use this table to initialize the three register stores. Those that are not explicitly initialized will be set to zero.

For example:

```
init
     fmpc := 25                 /* microprogram counter */
     local_store[25] := 100  /* start address        */
     external_store[8..10] := 2,4,8  /* constants    */
endinit
```

4.4 Execution Block

The execution block, as shown by Figs. 7 and 8, is made up solely of procedures. Each procedure may have as its first statements synonym declarations which rename variable identifiers locally for the duration of the procedure. The scope of a local synonym applies only to the procedure in which it is declared and does not extend to any called procedures. Fig. 10 illustrates the basic statement constructs and the reserved words associated with each. The approach in this section is to first explore assignment statements and tests, followed by the semantics of procedures. Branch, conditional, repetition, and parallel statements are then discussed followed by the new case

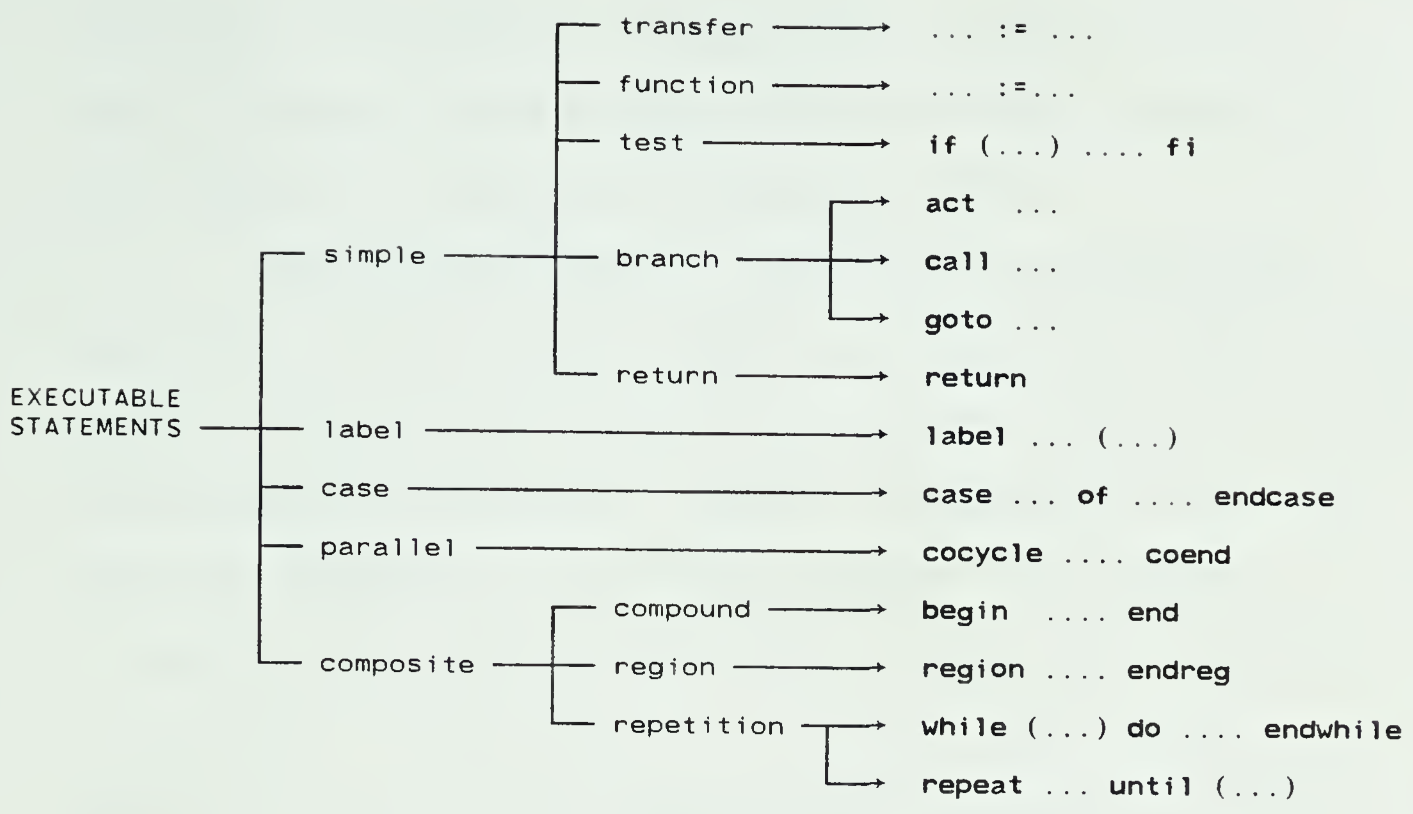


Figure 10. Executable Statements

construct.

4.4.1 Assignment Statements

4.4.1.1 Transfer Statements

The transfer statement *sink* := *source* is the simplest form of assignment and is used when there is a direct connection between the *sink* and *source* via a residual control register, 6-bit or 18-bit bus or through main or control memory. When the *sink* is a local store register and the source is an 18-bit bus, or a local store and external store register, the ':=' is simply an abstraction for the *gate nanoprimitive* associated with the transfer.

The use of an array pointer--a residual control register--indicates that no action other than the gating is

required, otherwise the appropriate residual control register must first be setup. This residual control setup also applies to function statements. Thus:

```
        local_store[fcod] := control_store_output_bus;
```

signifies to the compiler that the F-register FCOD already points to the proper local store register and only the *gate cs* nanoprimitive need be issued. The semantics for these types of transfers is straightforward. If a pointer is used it must be the correct one (there is only one) for the bus in question. If the *source* is a local store register and the *sink* is a bus only the appropriate F-register is setup as no gating is involved.

A memory reference to either main or control store is handled in the same manner except the memory array pointer will specify the address. Memory reads may be placed onto the appropriate data bus or gated directly into a local store register. A full main store read would look like:

```
    local_store[11] :=
        mainstore[mainstore_source[local_store[10]]];
```

or if the F-registers were previously set:

```
        local_store[fmod] := mainstore[fmix];
```

The main store write is slightly more complex as the MIX input bus multiplexes both the address and data. The address must first be established on the bus (controlled by the F-register FMIX) after which the data is supplied to the same bus by (possibly) changing the contents of FMIX to point to the local store register containing the data. For

example:

```
        main_store[fmix] := local_store[10];
```

will leave the value 10 in the F-register FMIX which may not have been there originally. For a full read or write operation from memory to a local store register the compiler will include the looping code necessary to test for memory ready and data ready. The programmer has control over this and may substitute his own code for looping if desired.

Several options are available to the programmer for accessing control store as shown in Fig. 5. Four increment operators: **+1**, **+2**, **+b** and **+ab** for use with the MPC unit have been incorporated. Thus, for example, to read **+b** words past the present location pointed to by the microprogram counter the following would be used:

```
    local_store[fcod] := control_store[index[fmpc] +b]
```

4.4.1.2 Function Statements

Function statements represent data transformations and have the form: *sink* := *expression* where *sink* is an output data bus, a local store register, or an F-store register. The expression may contain a maximum of two terms separated by a single operator which is limited to functions performed with one pass through an ALU or the shifter [SALI76]. To avoid ambiguities we have incorporated new reserved words into S*(QM-1) to specify QM-1 related operators.

For example, there are four distinct methods of

directly incrementing a register, denoted by the symbols: **incl**, **xincl**, and **+1** (representing two distinct increments, determined by context). The operator **incl** indicates that the left input of the ALU is to be incremented while **xincl** causes the left input to the index alu to be incremented. The **+1** operator is used for incrementing the 6-bit F-store registers and the four special local store registers that can serve as microprogram counters.

The effect of the residual control registers requires a strict ordering in two term ALU operations. The register to the left of the operator is the left input and the one on the right the right input. Thus the ordering for the ALU is:

```
    local_store[faod] := local_store[fail] <alu op>
        local_store[fair]
```

Index ALU operations (Fig. 6) are differentiated from the ALU by context where the *sink* is a local store register pointed to by *gspec* as in **local_store[g_store[3]]** which indicates that the third G-register (F-register 22) points to the local store register to receive the index ALU output. Ordering of expression terms is not as important as with the ALU, as the compiler is able to determine the inputs by context, but are retained for consistency.

An analysis of over 17,000 lines of source nanoassembler code (resulting in approximately 700 nanowords), taken primarily from Demco's PDP-11 emulator and MULTI nanocode [NANO75], was undertaken to choose which data

transformation operations should be identified by reserved words. For example, from a total of 32 different operations available for the ALU's, five--addition, pass left,. increment left, pass right, and subtraction--were used 80% of the time, and 17 operations were never used at all. The analysis resulted in forty reserved words being chosen from an approximate total of 135 operations.

In the event a nonreserved word operation is required, provision has been made for the programmer to use it at the expense of using a slightly lower level of programming. The operator may be specified indirectly as a constant or as a *k_vector_register*, enclosed in brackets to indicate that the appropriate value has been previously assigned to the K-vector field. This is also a method for introducing either *a* or *b* parameters of a control store instruction (through R31) by assigning the parameter to a k_vector_register before the operation is performed. In the case of the index ALU only constants may be used in this fashion with the index ALU as the operation must be encoded by the compiler into the *fsel2* field in the T-vector.

Several **pvar**'s are available for manipulating ALU carry controls and setting the global condition F-register, FIST, with the local conditions. Global condition setting is abstracted, by S*(QM-1), to occur after the ALU or shifter operation is completed as in:

```
        kalc := instruction_reg.b;
        local_store[5] := local_store[fail] (kalc)
```

```
          local_store[5];
       alu_status := 1;
```

The ALU operation is specified by *(kalc)* which indicates to the compiler that the 6-bit register specifying the ALU operation to be performed has been previously established. This allows the programmer to perform ALU operations indirectly, as shown in the example, or perform operations which are not directly supported by S*(QM-1).

Shifter operations are specified in the following order: type of shift (arithmetic, logical, and circular), direction (<< or >>), mode (double), a number representing the shift count, and finally the shifter input. A zero shift amount passes the input directly to the output. For example:

```
        local_store[fsod] := a<<10 local_store[10];
        local_store[1] := c>>d 3 local_store[fsid];
```

The first statement shifts the contents of the tenth local store register arithmetically left ten positions and gates the result into the local store register pointed to by the F-register FSOD. The second statement does a right circular shift of three positions using the data on the shifter input bus and the output of the alu (which forms the high order 18-bits) and gates the low order 18-bits (shifter output data) into the first local store register.

4.4.1.3 Test Expressions

The S*(QM-1) testing facility is used with if, **repeat**, and **while** statements. It allows programmers to test

single-bit conditions directly, and multi-bit conditions indirectly using either a constant, or a K-vector variable representing a mask register. Tests may be performed on reserved words, variables or expressions. The reserved words, such as **overflow**, and **result**, represent single-bit conditions which are tested against either zero or one. Associated with each of these tests is a modifier indicating which of either the local, global or special conditions are to be tested. If not present a local test is assumed unless a special condition may be unambiguously determined by compiler.

Variable and expression tests are either local or special depending upon the context. A variable test refers to an F-store register tested for zero (== 0) or not zero (!= 0). Two local store registers may be tested against each other (> !> <! == !=), which is accomplished by the compiler using a subtraction and a local test for zero. Expressions refer to ALU or index ALU operations (as in function statements) which are performed without gating the result into a local store register. Tests involving the ALU use local conditions and have the obvious side effect of changing the residual control registers.

4.4.2 Procedures

4.4.2.1 Instruction Procedures

The micro assembler written and used (under UNIX on a PDP-11/45) at the University of Alberta, which assembles control store microprograms based on the instruction set implemented in nanostore, requires that a "definition" file be available for assembling microprograms. The compiler must build this file based on the opcode it assigns to the microinstruction, and parameters in the procedure statement. Parameters indicate the mnemonic name (op =), which defaults to the procedure name, and the format (fmt =) of the instruction. For example, an instruction procedure may be defined in the following manner:

```
proc add_immediate (instruction, op=addi, fmt=r.r.c)
              ...
              ...
endproc
```

where the r's refer to local store registers and c is the immediate 18-bit value following the instruction.

Due to the layout of nanostore and the method of microinstruction invocation, as depicted in Fig. 11, the first nanoword is placed in a header page defined and set by the compiler. The subsequent words (if there are any) are placed in a separate page and are referred to as the body of the instruction.

The first word of each instruction will have a bit set which indicates that the instruction is a legal entry point

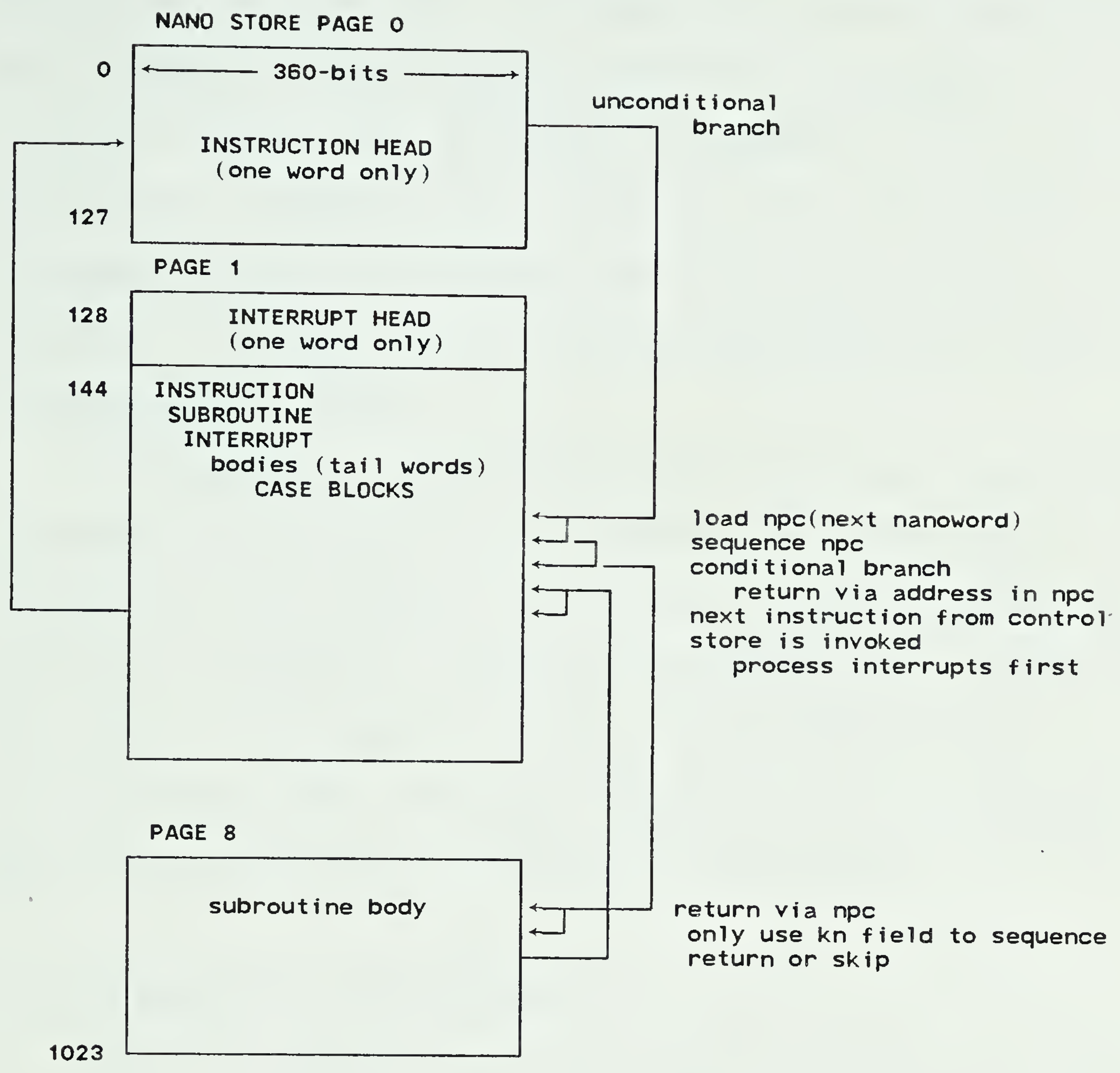

NANO STORE PAGE 0
0
360-bits
INSTRUCTION HEAD (one word only)
127
unconditional branch
PAGE 1
128
INTERRUPT HEAD (one word only)
144
INSTRUCTION SUBROUTINE INTERRUPT bodies (tail words) CASE BLOCKS
load npc(next nanoword)
sequence npc
conditional branch
return via address in npc
next instruction from control store is invoked
process interrupts first
PAGE 8
subroutine body
1023
return via npc
only use kn field to sequence
return or skip

Figure 11. S*(QM-1) Nanostore Layout

for the invocation of a microinstruction. If an illegal instruction is fetched it will point to a nanoword in the instruction page which is not an entry point, causing a program interrupt.

The first word branches directly from the header page to the first word of the body of the instruction which is located in one of the remaining pages allocated by the compiler on a first-come-first-placed basis. Then the

nanoprogram counter, NPC, is loaded with the address of the next nanoword from the address field in the present nanoword. Control in a straight line segment (i.e. a sequence of statements with one entry point--the first statement, one exit point--the last statement, and no branches in between) is then accomplished, by the compiler, by sequencing the NPC.

Interrupts are permitted to occur after an instruction procedure, by setting the permission bits in the last nanoword of the procedure. The "return" from the interrupt routine is effected by simply using the NPC as a pointer to the start of the next instruction. This implies that interrupts use only the branch capability in each nanoword and do not alter the NPC contents in any manner.

4.4.2.2 Subroutine Procedures

The QM-1 supports only one subroutine level with the return address being held in the NPC which must remain unaltered if the subroutine wishes to successfully return from whence it was called. Control (generated by the compiler) within a subroutine is passed from one word to the next by setting the *next address field* to the address of the next word and setting the branch bit in the nanoword. The nanoprimitive *read ns* will cause the word specified by the branch address field to be read. When the *gate ns* nanoprimitive is issued this word will be gated into the *control matrix*.

Subroutines are defined as follows:

```
proc procedure_name (subroutine, allow_interrupts)
             ...
             ...
endproc
```

where the parameter allow_interrupts is optional and if used indicates to the compiler, that upon completion of the subroutine, interrupts may be taken.

The method used to conditionally return to the calling instruction procedure requires that the branch address be set to the next word. The branch and alternate branch bits must also be set to one. The word specified by the branch address is read by a *read ns* nanoprimitive, and a condition test is made to see if a *gate ns* should be performed. If the next word is gated, control remains in the subroutine. On the other hand, if the gate does not occur, the branch bit is set to zero, and on the next *read ns* the NPC is used to specify which word in nanostore should be read. This is of course the return address. An unconditional return is accomplished by setting the branch bit to zero and using the NPC as the return address.

4.4.2.3 Interrupt Procedures

Interrupt procedures are defined in the following manner:

```
proc procedure_name (interrupt, level = ... )
             ...
             ...
endproc
```

The second parameter refers to the interrupt(s) the procedure is to handle. Interrupts are permitted to occur on completion of a microinstruction after the NPC has been loaded with the address of the next instruction. Only branch addressing is employed by the compiler within interrupt procedures, thereby preserving the NPC for activation of the next instruction. Interrupt addresses are kept in external store in a compressed form which allows the header nanowords to appear in the first sixteen words of the first four pages. Based on a parameter indicating which of up to 30 interrupts are to be handled, the compiler will allocate header and body nanowords, set enabling bits, and prepare interrupt addresses for external store in compressed form.

4.4.3 Transfer of Control Statements

The *call* statement, **call** <name> (where <name> identifies a subroutine procedure), may only be used in an instruction procedure. It invokes a subroutine with the implication that a return will be made to the statement following the call. The return is either specified directly via the **return** reserved word or is implied by falling through the bottom of the subroutine procedure. The *activate* statement, **act** <name>, is a non-return call which is essentially a **goto** from inside a procedure to the beginning of a subroutine or another instruction procedure.

4.4.4 Conditional Statements

A conditional **if** statement is of the form:

if(test expression) ... fi

where one or more statements (the body) represented by the ellipses ("...") are executed if the test condition is true. The compiler ensures that sequencing within the body of an **if** statement is done using the NPC. For the case where the test condition is false the compiler generates code to cause a branch to the statement following the **if** statement.

As indicated previously a subroutine procedure must not alter the NPC if a successful return to the calling procedure is to be made. This implies that a subroutine may not call a procedure. Also, the body of the **if** statement may not use the NPC for sequencing. Therefore only a **return** statement which causes control to be returned to the calling instruction procedure or a **cocycle** statement (see section 2.5.1) which uses only the *skip* nanoprimitive may be used.

The repeat statement, as shown in Fig. 10, is provided as an abstraction for single nanoword loops, which take advantage of the cycling effect of the active nanoword. Statements within the repeat statement must be *compactable* into one nanoword and therefore may only be assignment, **cocyle** or **region** statements. The compiler supplies code such that if the test expression is true a conditional *gate ns* occurs transferring control to the next statement following the repeat. A false test result causes the *gate ns* to not

occur resulting in the next T-vector, T1, becoming active. Repeat statements may be used in any of the three types of procedures.

The **while** statement is a general purpose repetition statement providing a multi-nanoword looping capability. The body of the while statement is supplied with sequencing code which uses the NPC. This implies that these statements may only be used in instruction procedures. If the test expression is false a branch using the *next address field* causes control to be passed to the statement following **endwhile**.

4.4.5 Parallel Statements

As Fig. 10 indicates, the only parallel statement used in S*(QM-1) is the **cocycle** statement. Although the QM-1 employs a three-clock timing scheme all micro operations, from the point of view of an S*(QM-1) programmer, may be considered to be executed in parallel. Thus statements within a **cocycle** statement may only be separated by the '□' symbol indicating the parallel nature of the statements.

The stcycle statement as discussed in the second chapter is not applicable as there are no nanooperations which may begin executing at the same time but complete at a different time.

4.4.6 Case Statement

The **case** statement is a new construct offering an order

dependent multi-way branch facility. It takes advantage of the fact that an illegal byte specification when writing to nanostore causes only a read to occur (see section 3.2). Because of the intricacies involved in setting up nanostore to perform this multi-way branch, the programmer must declare within the declaration block the name of the **case** statement and the number of elements it will contain. This allows the compiler to select a base address (in the *tail* word section of nanostore) and reserve the required number of nanowords. Each of these words performs the same action as instruction header words causing a branch to another section of nanocode where the associated routine may be completed. Since instructions and interrupts are assigned specific header word locations in nanostore only subroutines and label statements may be used in a case statement. The subroutines and labels must be explicitly identified with the case statement using a parameter: **case** = <case name>. As the compiler detects these parameters it puts the header word of the routine in the next available position in the case block, constructs a branch and places the remainder of the routine in a free section of nanostore. These routines are entered from the **case** statement using either **call**, **act** or **goto**.

All the programmer needs to do is supply the offset, which for convenience will be in the B field of local_store[31] (R31). The base address is supplied by the compiler and the offset is moved to the A field completing

the 10-bit nanostore address. The compiler supplies an illegal byte address and the specified word is "read" from nanostore.

Provision has been made in the **case** statement to allow the programmer to make an assignment (a transfer statement not including memory reference) to R31 after its contents have been used for the address but before the word read is actually gated into the control matrix. The following is a short example:

```
        case  jump_table  : 4 /* in declaration block */
        case  jump_table  of
              0  :  call subroutine_one
              1  :  goto label_two
              2  :  act instruction_three
           local_store[31] := alu_output_bus
        endcase;
```

The label and subroutine names must appear in the program in the order that they appear in the case statement. The integer serves to indicate the offset and must be consecutively numbered, beginning with zero.

Chapter 5
The S*(QM-1) Compiler

The purpose of this chapter is to explore the issues and design aspects related to the development of a compiler (first phase) for the translation of S*(QM-1) programs into a sequential intermediate language. The second phase addresses the problem of compacting this intermediate language code into efficient and correct QM-1 nano-object code and is being studied independently [RIDE81a]. The intermediate language and associated semantics were developed jointly and reflect the constraints imposed on the compactor.

The immediate goal in the design of a compiler was to provide support for the instantiation of S* to S*(QM-1), as detailed in the second chapter. We felt that a quickly implemented parser capable of syntactic checking, error reporting and recovery would be a definite asset. Towards this end a preprocessor was written, and the UNIX tools LEX and YACC were used, with support programs, to produce a working parser. This is quite extensible and the remaining work may easily be constructed within the framework developed. We shall briefly consider the functions performed by the parser and then examine the development of the intermediate language. Finally, the design aspects which the present compiler will have to take into account to translate S*(QM-1) programs into the intermediate language will be

discussed.

5.1 A Partial Compiler

LEX generates a program, lex.yy.c, to perform lexical analysis of an input stream based on its input specification language (rules) which are in the form of regular expressions. From these expressions, transition tables are constructed defining a finite automaton which is then interpreted by an "included" C program.

The input stream is segmented into tokens based on the rules such that when a string is recognized a C program fragment associated with the regular expression is executed. Typically the program lex.yy.c is used by a parsing program to identify the next token in the input stream. The program fragment will then return a value associated with the token.

Once a string based on a regular expression has been identified LEX allows the programmer to put the characters back into the input stream for more analysis, if desired. Precedence rules make the identification of reserved words a simple matter so that they may be easily distinguished from symbol names.

YACC is also a program generator and takes as its input specification language a context free grammar describing the programmer's desired syntax. From this it constructs several parsing action tables which along with the included C program *yaccpar* perform the task of an LR(1) parser.

The basic action is to lookahead one terminal symbol, or token, from a user supplied lexical analyzer (in our case lex.yy.c), scanning from left to right, and construct a rightmost derivation in reverse. Based on the action tables these terminal symbols are shifted onto the parsing stack until a production rule is recognized. When this occurs user supplied C program segments associated with the production rule are executed, typically performing such actions as semantic checking, symbol table management, or intermediate code generation. This action is known as syntax directed translation.

Upon completion of the program segment the reduction is carried out by popping the associated terminal symbols off the stack. A check is made via the parsing action tables to see if a further reduction can be made based upon the newly exposed symbol on the stack and the nonterminal associated with the last reduction. If no reduction is possible the nonterminal symbol is shifted onto the stack. This process of shifting and reducing is repeated until either an error occurs, or the input string, the source program, is reduced to the start symbol of the grammar.

When used together the two programs lex.yy.c and y.tab.c communicate via tokens, the values of which are determined by YACC and included by LEX into lex.yy.c. These tokens are conceptually similar to the reserved words, although several cases arise when a group of operators are represented by one token.

When an error is detected by the parser it executes a user supplied routine which reports the line number and file name in which the error occurs. The parser then attempts to recover by discarding terminal symbols from its stack until it enters a state where the special token *error* is legal. At this point it attempts to resume its parsing action by looking ahead for three consecutive tokens which might legally follow the production rule with the *error* token. If this is accomplished it considers the error recovery to be successful and continues on as normal, otherwise it terminates.

The included file *yaccpar*, which interprets the tables produced by YACC, was modified to increase its parsing diagnostic reporting. It now provides more information on syntax errors, either from a program error or an incorrect grammar rule, so that they might be found quickly and corrected with a minimum of trouble. This is a compiler option under the control of the user with the use of several compiler flags and the reserved words **#debugon#** and **#debugoff#**.

A flexible preprocessor has been written to aid in developing and understanding S*(QM-1) programs. The use of the macro and file inclusion capability provides information hiding and chunking to allow programming details to be suppressed if desired. Macros must be declared within the declaration block using the format:

macro <identifier> .... **endmac**

The convention established is that the identifier should be in upper case (explaining why only lower case is permitted in S*(QM-1)). No restrictions have been imposed on the length of either the the body of the macro or its identifier. In order to be replaced the macro identifier must be the first non-white character field within a line.

The **#include** "file name" directive causes the preprocessor to replace the line with the file identified by file name. Macros within the included file are entered in the macro table or replaced as the circumstances dictate.

To aid in the correct incorporation of changes and the maintenance of interrelated source and object programs (of which the compiler is composed) a UNIX utility program, MAKE [FELD79], was employed. It uses a programmer supplied file dependency graph to perform actions, such as compiling or linking, after a source file has been altered. Combined with these specifications are user supplied programs which ensure that reserved word changes in either the lexical analyzer or parser are made correctly and consistently.

5.2 Translation and the Intermediate Language

It must be understood from the outset that the translator and compactor perform two entirely unrelated functions communicating in only one direction via the intermediate language. The language is designed around the nanooperations (NO's) and related fields within the QM-1 nanoword. Each NO is represented by a positive integer and

may have up to two parameters, primary and secondary, assigned to it. The compactor also treats these parameters as NO's, in their own right, which have a primary field consisting of the value the field is to contain.

For example, the NO *gate ns* has two parameters representing the test specifier and the associated K-vector mask value. A complete list of these NO's and related parameters may be found in the Appendix to Rideout's thesis on nanocode compaction [RIDE81b].

The translator thus transforms S*(QM-1) programs into a sequential list of NO's which the compactor *maps* [DEWI76b] into the fewest number of nanowords possible. The compactor maps individual portions of this sequential list called straight line microcode (nanocode) segments, SLM's, into nanowords (local compaction) and does not concern itself with the interaction between SLM's (global compaction) or optimization of the intermediate language. It attempts to produce the most efficient code possible but will, in some situations, sacrifice optimality of compaction to ensure correct code.

The translator must include with the NO's, control operators, represented by negative integers with up to two parameters, which partition the sequential list into SLM's. Additional information is also supplied informing the compactor of the type of nanoword sequencing required and positioning of nanowords within nanostore. Eleven control operators are presently being used:

1) start of SLM
2) force new nanoword
3) force new instruction (name)
4) force new interrupt (name)
5) force new subroutine (name) (case name =)
6) label (name) (case name =)
7) start of a region
8) end of a region
9) start of a cocycle
10) end of a cocycle
11) end of program

From this list of operators it is quite evident that the translator knows nothing about the relationships of statements within any of the parallel or composite statements. Its function is only in the domain of syntax and semantic error checking such as determining that it is incorrect to assign the output of the alu to the input of the shifter or to transfer the contents of an F-register to and external store register. It is unable to determine if the statements, for example within a **cocycle**, are data dependent and therefore cannot be executed in parallel, or if the semantically correct simple statements within a **repeat** are capable of being mapped into one nanoword. Thus a new type of error arises with compilation of S*(QM-1) programs which depends upon the compactor's ability to map NO's into nanowords depending upon timing, data dependencies

and resource conflicts. We call this a *mapping* error to distinguish it from syntax and semantic errors.

When a mapping error is detected the compactor is able to inform the programmer of the fact that an error occurred but is unable to supply any additional information except a dump of its packed words. It is up to the programmer to determine where the mapping error lies, and how to correct it, based on a knowledge of the QM-1 and the areas where the compactor might sacrifice optimality of compaction to ensure correct code.

5.2.1 The Semantics of Translation

The translation process of S*(QM-1) programs to a sequential list of NO's is straightforward requiring no decision making capabilities, such as which ALU to use or which data path to select, on the part of the translator. Essentially, a statement may have only one representation, although this may be expanded if the parameters are treated as NO's, which is obtained using a "brute force" approach. A convention has been established for transferring constants to F-register variables. The compactor employs a form of version shuffling [RIDE81a] to determine which of the K-vector constant fields should be used for the transfer. The translator passes only the F-register and the constant values to be assigned and lets the compactor perform the actual assignment. This is the only situation where the programmer does not have absolute control over the QM-1. The

advantage is that the programmer is freed from unnecessary details which the compactor is easily and efficiently able to handle. Should the programmer wish to circumvent this he may do so by explicitly assigning the constant to a k_vector_register variable and then assigning that variable to the F-register in question.

Most of the control operators are straight forward in their usage. Whenever a procedure is declared the appropriate operator with the procedure name is sent informing the compactor that it must start a new nanoword (with sequencing and nanoword placement as required) which is, of course, also the start of an SLM. The compactor keeps track of the location of the new nanoword and procedure name (label) associated with it for later transfer of control situations.

Labels force the start of an SLM and a new nanoword. If either labels or subroutine procedures have a case parameter the first nanoword is entered into the appropriate case block. **Region** and **cocycle** statements, which are not the start of SLM's, need control operators to ensure that the statements within are packed and executed correctly (see section 2.3.3.5). The complex field encoding required by an index ALU operation requires that the translator places the NO's between cocycle operators to ensure that the compactor does not separate them.

On completion of a procedure the compiler includes several NO's depending upon the semantics of the procedure.

For instructions the NO *allow interrupts* is passed unless an **act** statement proceeded the **endproc**. If the sequence: load npc(cs), load R31 and read cs (for instruction lookahead) is received by the compactor it ensures that the load R31 and read cs are executed in T-vector(s) following the load npc(cs). At the end of a subroutine or interrupt procedure an unconditional *gate ns* NO is used to force the use of the NPC (unless an **act** statement appears before the endproc) and the NO *allow interrupts* will be given, if the programmer included the parameter in the procedure head.

A **call**, **act** or **goto** branch statement requires that the NO *branch* (label) be given. In the case of **goto** a *load npc* (kn) nanoprimitive is used to correctly sequence the NPC to the statement that is being branched to.

If statements require the translation of the test expression into a 6-bit field (field) and the determination of which of the three K-vector masks (test specifier) is to be used. The NO *prep branch* (label) and *gate ns* (test specifier) (field) operators are issued which causes the NPC to be used if the test expression is true and a branch to label if false. The label name is generated by the translator and refers to the statement following the fi. A *start of SLM* is issued followed by the NO's of the statements to be executed on a true expression. When the fi is detected the translator inserts a label control operator with the name generated above which forces a new nanoword and starts a new SLM.

If the only statement in the body of an **if** statement is a **cocycle** the *skip* nanooperation is used which skips over the T-vector with the **cocycle** statement. The NO, *skip* (test specifier) (field), is passed where the two parameters are determined as above, such that the test specifier is the reverse of the mask required so that a *skip* will take place if the test expression is false. Recall that the NPC may not be used within a subroutine or interrupt procedure (it is needed to hold the return address) so that only **cocycle** or **return** statements may be used in the body of an **if** statement. In this case no label need be generated, since a branch will be taken to the next nanoword (handled by the compactor) if the test expression is false.

Repeat statements must be compactable into one nanoword, may include only transfer, function, **cocycle** or **region** statements, and may be used in any type of procedure. The translator passes the control operator *force new nanoword* on detection of **repeat** which is followed by the NO's making up the statements in the body. The test expression is translated and *gate ns* (test specifier) (field) is given. If the test expression is true the word last read form nanostore is gated into the control matrix, otherwise the next T-vector in the present word in the control matrix is activated.

The **while** statement: **while** (test expression) .... **endwhile** is expanded by the compiler to:

```
        label : name1
```

```
if (test expression)
   ....
   ....
   goto name1
fi;
label : name2
```

where a branch will be executed to name2 if the test expression is false. The while statement may only be used in instruction procedures due to the NPC requirement given with if statements.

Chapter 6

Conclusions

This thesis has provided a wide ranging examination of the high level microprogramming language schema S*. We have introduced S* in the overall context of the family of design languages [S*] and described an example of its use in the development of a language directed architecture. The process of instantiation has been discussed at length, and we have provided the first critical survey of S* from which several recommendations have been made. The true test of S* has come, however, with its instantiation to S*(QM-1). We feel that the language schema and its underlying philosophy has met the challenge presented by the nano-architecture of the QM-1.

Our version of S*(QM-1) is a machine dependent high level microprogramming language which presents a novel procedural approach to nanoprogramming the QM-1. It functions in harmony with the nano-architecture and fulfills the QM-1's designers' [ROSI79] intentions of developing indirect emulations. We have mapped S* constructs effectively onto the nano-architecture providing a high level abstraction of the QM-1 which does not sacrifice the programmers control over the hardware.

The ground rules established by Dasgupta for high level microprogramming languages have proved important in the development of S*(QM-1). Without the parallel and sequential

flow of control constructs (**cocyle** and **region**) it is doubtful if the compactor would be able to always guarantee the correct production of nano-object code due to the effects of residual control and the complex encoding frequently required within nanowords. The importance of having S*(QM-1) programs independent of the nanostore organization is clear when the complex flow of control between nanowords and the method of control store instruction interpretation are considered. Finally, the data structuring capabilities provided by S* allows the declaration of variables which are meaningful and representative of the architecture of the QM-1.

The question that must be answered is whether or not S*(QM-1) is an effective and efficient tool for nanoprogramming the QM-1. There can be no doubt about the advantages of a well structured high level language over assembler, especially nanoassembler, programming. S*(QM-1) provides the capability of writing well structured programs which allow the programmer to employ a hierarchical top down design approach for the implementation of his algorithms. This was clearly established by Olafsson in the writing of the instruction interpreter for the QMC [OLAF81] which took less than two weeks to transform from an S*A description to an S*(QM-1) program.

S*(QM-1) programs are, by their very nature, more reliable than their nanoassembler counterparts as many minute details of nanoprogramming are taken care of by the

compiler. This was illustrated with the writing of a subroutine for the mode calculations [KLAS81] required for a PDP-11 emulation based on the work by Demco [DEMC76]. In one particular mode Demco had inadvertently omitted an important masking operation required to ensure that only a 16-bit address was used for fetching an operand. This was a well hidden bug which would have seldom caused any problem (it had lain undiscovered for five years) and was not found because of the amount of detailed code surrounding it. Its existence became immediately apparent when the two programs were compared and had it happened in the S*(QM-1) program it would have stood out to a much greater extent that in the nanoassembler program.

The argument may be presented that an effective microprogramming tool without efficient code generation is no tool at all. Recent tests of the compactor have demonstrated between 75-80% efficiency in its packing compared to an assembler programmer. With an optimizer and global compaction this figure could be substantially improved upon. In this form the use of S*(QM-1) easily outweighs, in terms of ease of program implementation and improved reliability, any speedup which may be gained by using the nanoassembler.

6.1 Recommendations

The following recommendations are made in regards to S*, S*(QM-1) and the QM-1:

1. The changes of section 2.3.3 should be incorporated into S* based on their use in S*(QM-1). These include new reserved words, expanded method of variable declaration, several new constructs, the grouping together of transfer of control statements into a new executable statement type--branch statements, incorporation of the **act** statement, and the use of ';' only with executable statements.
2. A new construct **init** ... **endinit** should be incorporated for the initialization of machine registers before the execution of the program begins.
3. A **case** statement offering a method of multi-way branching involving position dependent statements is applicable and should be incorporated.
4. Due to the effects of residual control, situations arise when the compactor is forced to produce less than optimal code to ensure its correctness. If an output data bus is gated (into local store) prior to a mainstore or control store (with CIA) operation the compactor is forced to assume that an address is being generated with the gate and may not pack the memory operation and gate together. In the case of an index ALU operation following a read control store operation the compactor must assume that COD will be the left input

into the index ALU and will therefore not pack the gate and index operation together.

To alleviate this problem a new construct is proposed:

parbegin S1 // S2 // S3 **parend**; which states that the statements S1, S2 and S3 are *data independent* and may be compacted as optimally as possible. With this in mind the compactor would even be able to rearrange the order of the statements (a form of exhaustive enumeration) to obtain the most optimal packing possible.

5. One of the major drawbacks with S*(QM-1) results from the inability to save the NPC. This prevents subroutines calling other subroutines or the use of **while** statements in subroutines. The statements allowed within an **if** statement and in subroutines, are also severly limited because of this restriction. These problems could be solved easily with an *NPC stack*. Pushing and popping could be performed using an *auxiliary action* nanooperation and two unused command specifiers for the special F-register FACT. New circuitry and stack hardware would have to be added but the problem does not seem insurmountable compared to the advantages which would be gained.

6.2 Future Work

S*(QM-1) has been instantiated, a compiler has been designed, and the compactor has been implemented. The two compiler phases must now be brought together and a working

compiler demonstrated. The development of a working compiler, presents several practical research problems. A global optimizing strategy to detect potential concurrency in the intermediated language code should be developed. It should also be able to recognize and limit the number of residual control transfers required. In its present implementation an S*(QM-1) programmer is well advised to set up his F-registers as infrequently as possible to limit the number of transfers required. This is considered to be a flexibility of S*(QM-1) as a previously setup register may be referred to, for example, as **local_store[feod]** or **local_store[fair]**. To understand exactly which local store register is being referred to requires the reader to backtrack to where it was setup. A more reasonable approach would be to always specify the subscript and let the optimizer determined when the respective F-register should be allocated.

Once experience is gained with a working compiler heuristics may be built into it to relieve the programmer of always having to specify every action explicitly. For example, a semantically reasonable statement should be **local_store[6] := local_store[7]** where the data path is left unspecified and is determined by the compiler. Increments and decrements could be treated in the same manner where it would be the compiler's responsibility to determine which ALU is the most effective to use. These types of operations would require a detailed data base of operations which come

before and after the statement in question. With this knowledge a decision involving the most efficient data path or transformer to use should be possible.

More instantiations should be performed, especially on bit slice architectures, to provide further experimental evidence for the use of S*. The tools that have been developed for this research should apply to further instantiations, aiding in the design and implementation of instantiated languages.

Bibliography

[AGRA76] Agrawala, Ashok K., and Tomlinson G. Rausher. *Foundations of Microprogramming--Architecture, Software and Applications.* New York, N.Y.: Academic Press, 1976.

[AHO79] Aho, Alfred V. and Jeffrey D. Ullman. *Principles of Compiler Design.* Don Mills, Ont.: Addison-Welsey, 1979.

[DASG76] Dasgupta, Subrata. "Parallelism in Microprogramming Systems". Diss. University of Alberta, 1976.

[DASG78] Dasgupta, Subrata. "Towards a Microprogramming Language Schema." *Proc. 11th Annual Workshop on Microprogramming (MICRO-11),* Nov. 1978, Pacific Grove, CA., pp. 144-153.

[DASG80a] Dasgupta, Subrata. "Some Implications of Programming Methodology for Microprogramming Language Design." *Microprogramming, Firmware and Restructurable Hardware,* Ed. Gerhart Chroust and Jorg Mulbacher, North-Holland Amsterdam, 1980.

[DASG80b] Dasgupta, Subrata. "Some Aspects of High Level Microprogramming." *ACM Computing Surveys,* Vol. 12, No. 3 (1980), pp. 295-323.

[DASG81a] Dasgupta, Subrata, and Marius Olafsson. "Towards a Family of Languages for The Design and Implementation of Machine Architectures." *Technical Report TR81-5,* University of Alberta, Edmonton Alberta, T6G-1H7, June 1981.

[DASG81b] Dasgupta, Subrata. *Private Communications.* 1981.

[DAVI78] Davidson, Scott and Bruce D. Shriver. "An Overview of Firmware Engineering." *IEEE Computer,* Vol. 11, No. 5 (1978), pp. 21-33.

[DEMC76] Demco, John C., and Tony A. Marsland. "An Insight into PDP-11 Emulation." *Proc. 9th Annual Workshop on*

Microprogramming (MICRO-9), Sept. 1976, pp. 20-26.

[DEWI76a] DeWitt, David J. "Extensibility--A New Approach for Designing Machine Independent Microprogramming Languages." *Proc. 9th Annual Workshop on Microprogramming (MICRO-9)*, Sept. 1976, pp. 33-41.

[DEWI76b] DeWitt, David J. "A Machine Independent Approach to the Production of Optimized Horizontal Microcode". *Technical Report 76 DT 4*, University of Michigan, August 1976.

[ECKH71] Eckhouse, R.H. "A High-level Microprogramming Language (MPL)." *AFIPS Conference Proceedings, 1971 Spring Joint Computer Conference*, Vol. 38 (1971), pp. 169-177.

[FELD79] Feldman, Stuart, I. "Make--A Program for Maintaining Computer Programs." *Software--Practice and Experience*, Vol. 9 (1979), pp. 255-265.

[FLYN71] Flynn, Michael J., and Robert F. Rosin. "Microprogramming: An Introduction and Viewpoint." *IEEE Transactions on Computers*, Vol. C-20, No. 7, 1971, pp. 727-731.

[JOHN80] Johnson, Stephen, C. "Language Development Tools on the Unix System." *Computer*, Vol. 13, No. 8 (1980), pp. 16-21.

[JOHN79] Johnson, Stephen, C. "A 32-Bit Processor Design." *Computing Science Technical Report #80*, Murray Hill, New Jersey: Bell Telephone Laboratories, Apr. 2, 1979.

[KERN81] Kernighan, Brian W., and John R. Mashey. "The UNIX Programming Environment". *Computer*, Vol. 14, No. 4 (1981), pp.12-24.

[KLAS81] Klassen, Alynn B. and Subrata Dasgupta. "Syntax and Semantics of the High Level Microprogramming Language S*(QM-1)." *Technical Report TR81-3*, University of Alberta, Edmonton Alberta, T6G-1H7, July 1981.

[LAND80] Landskov, David, Scott Davidson, Bruce Shriver and

Patrick W. Mallett. "Local Microcode Compaction Techniques." *ACM Computing Surveys*, Vol. 12, No. 3 (1980), pp. 261-294.

[LESK79] Lesk, M. E. "Lex--A Lexical Analyzer Generator." *UNIX Programmers Manual*, Seventh Edition, Vol. 2B, Section 20, Murray Hill, New Jersey: Bell Telephone Laboratories, 1979.

[MALI78] Malik, Kamran and Ted Lewis. "Design Objectives for High Level Microprogramming Languages." *Proc 11th Annual Workshop on Microprogramming (MICRO-11)*, Nov. 1978, Pacific Grove, CA., pp. 154-160.

[NANO75] Nanodata Corporation. *MULTI--Nanocode Segment.* July 1975.

[NANO79] Nanodata Corporation. *QM-1 Hardware Users Manual.* Third Edition, Revision 1, Buffalo, New York: Nanodata Corporation, 1979.

[NASH79] Nash, James and Mike Spak. "Hardware and Software Tools for the Development of a Micro-Programmed Microprocessor." *Proc. 12th Annual Workshop on Microprogramming (MICRO-12)*, Nov. 1979, Hershey, Penn., pp. 73-83.

[OLAF81] Olafsson, Marius. "The QMC: A Microprogrammed Instruction-Set Architecture." *M.Sc. Thesis*, University of Alberta, 1981.

[PATT76] Patterson, David A. "Strum:Structured Microprogram Development System For Correct Firmware. *IEEE Transactions on Computers.* Vol. C-25, No. 10, 1976, pp. 974-985.

[RAMA74] Ramamoorthy, C.V., and M. Tsuchiya. "A High Level Language for Horizontal Microprogramming." *IEEE Transactions on Computers*, Vol. C-23, No. 8 (1974), pp. 791-801.

[RIDE81a] Rideout, Douglas. "Considerations for Local Compaction of Nanocode for the Nanodata QM-1." *Technical Report TR81-7*, University of Alberta, Edmonton Alberta, T6G-1H7, June 1981.

[RIDE81b] Rideout, Douglas. "An Application of a Microcode Compaction Technique to the Nanodata QM-1 Nano-architecture". *M.Sc. Thesis.* University of Alberta, 1981.

[ROSI69] Rosin, Robert F. "Contemporary Concepts of Microprogramming and Emulation." *ACM Computing Surveys,* Vol. 1, No. 4 (1969), pp. 197-212.

[ROSI72] Rosin, Robert, Gideon Frieder, and Richard H. Eckhouse Jr. "An Environment for Research in Microprogramming and Emulation." *Communications of ACM,* Vol. 15, No. 8 (1972), pp. 748-760.

[SALI76] Salisbury, Alan B. *Microprogrammable Computer Architectures.* New York, N.Y.: American Elsevier Publishing Co., Inc., 1976.

[SINT80] Sint, Marleen. "A Survey of High Level Microprogramming Languages." *Proc. 13th Annual Workshop on Microprogramming (MICRO-13),* Dec. 1980, Colorado Springs, Colo., pp. 141-153.

[TOKO78] Tokoro, Mario, T. Takizuka, E. Tamura, and I. Yamaura. "A Technique of Global Optimization of Microprograms." *Proc. 11th Annual Workshop on Microprogramming (MICRO-11),* Nov. 1978, Pacific Grove, CA., pp. 41-50.

APPENDIX 1. SYNTAX AND SEMANTICS OF S*(QM-1)

1. Introduction

The purpose of this appendix is to provide the syntax and semantics of the high level microprogramming language S*(QM-1) which is used for *nanoprogramming* the Nanodata QM-1 [NANO79, SALI76, AGRA76]. The language has been designed to support indirect emulation [ROSI72] and provides an abstraction of the nano-architecture of the QM-1. Since S*(QM-1) is tied so closely to the QM-1's hardware it is strongly recommended that the QM-1 Hardware User's Manual [NANO79] be consulted before this appendix is studied.

2. Notation, Terminology and Vocabulary

Syntactic constructs are denoted by English words enclosed between the angular brackets < and >. The metanotation {x} refers to zero or more repetitions of the entity x. The meta-notation ↓y↓ refers to none or one instance of the entity y.

The basic vocabulary :

```
<empty> ::=

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|
               o|p|q|r|s|t|u|v|w|x|y|z

<digit>  ::= 0|1|2|3|4|5|6|7|8|9

<special symbol> ::= [ | ] ( | ) | , | ; | . | □ | := |
                      = | : | + | _ | - | +1 | +2 | +b |
                      +ab | > | == | < | !> | != | !< |
                      a<<↓d↓ | l<<↓d↓ | c<<↓d↓ |
                      a>>↓d↓ | l>>↓d↓ | c>>↓d↓ |
                      prog | endprog | declaration |
                      endec| tuple | endtup | init |
                      endinit | proc | endproc | if |
                      fi | cocycle | coend | begin |
                      end | region | endreg | case |
                      of | endcase | repeat | until |
                      while | do | endwhile | type |
                      seq | bit | var | pvar | syn |
                      array | with | const | bin |
                      oct | dec | hex | instruction |
                      subroutine | interrupt | label |
                      goto | act | call | return |
                      ↓x↓ones | ↓x↓zero |
                      ↓x↓notl | ↓x↓passl | ↓x↓incl |
                      ↓x↓decl | ↓x↓notr | ↓x↓passr |
                      xor | and | nand | or | nor |
                      special | local | global |
                      overflow | carry | ms_busy |
                      ms_data | slb | slb | r_index |
                      result
```

Upper case letters are not used in S*(QM-1). They have been reserved for use with macros, which is a compiler

(preprocessor) implementation feature and will not be discussed here.

Throughout this appendix special symbols are highlighted in **bold** print.

3. Identifiers and Numbers

Identifiers are the basic entities used to denote variables, constants, types, procedures, and labels.

```
<identifier> ::= <letter> | <identifier><letter> |
                  <identifier><digit> |
                  <identifier>_<identifier>



<bin digit> ::= 0 | 1
<oct digit> ::= <bin digit> | 2 | 3 | 4 | 5 | 6 | 7
<dec digit> ::= <digit>
<hex digit> ::= <digit> | A | B | C | D | E | F

<radix> ::= bin | oct | dec

<bin integer> ::= bin <bin digit>{ <bin digit>}
<oct integer> ::= oct <oct digit>{ <oct digit>}
<dec integer> ::= ↓dec↓ <dec digit>{ <dec digit>}
<hex integer> ::= hex <hex digit>{ <hex digit>}
<integer>     ::= <bin integer> | <oct integer> |
                   <dec integer> | <hex integer>
```

An identifier may be any length and must begin with a letter. This may then be followed by any combination of letters or digits separated by a single underscore. For example:

```
local_store
ax3py_a9
control_store_address_source
```

Except for the initialization block (to be discussed later) integers may only be assigned to an f_register or k_vector_register implying they must have a value less than 63 (2^6-1).

4. Program Specification

```
<program> ::= prog ( <identifier> ) <declaration block>
                <initialization block> <execution block>
                endprog
```

An S*(QM-1) program consists of a declaration block, an initialization block, and an execution block, and is uniquely identified by <identifier>. The major component of the declaration block is the pre-defined data objects (Appendix 3) which are global in scope and serve to explicitly define all data objects to which an S*(QM-1) programmer has access to. Any pre-defined variable data object may be locally or globally renamed with the synonym construct to suit the programmer's application. Also within the declaration block constants may be defined but must lie within the range 0 - (2^6-1). Case declarations are required for use with case statements due to the idiosyncrasies of the QM-1 nanostore organization.

The initialization block serves to furnish information to the QM-1 loader so that QM-1 registers may be initialized before emulation begins. The execution block consists of one or more procedures each of which may contain local synonym declarations and one or more executable statements.

5. Declaration Block

```
<declaration block> ::= declaration <decln list> endec

<decln list> ::= <data object> {<decln list>} |
                  <syn decln> {<decln list>} |
                  <const decln> {<decln list>} |
                  <case decln> {<decln list>}
```

QM-1 hardware resources are abstracted in terms of data objects. These data objects are pre-defined and explicitly delineate the totality of resources to which an S*(QM-1) programmer has access to and the manner in which they may be accessed. Since the data objects are pre-defined the programmer does not have to write his own declarations, he only has to know how to correctly use them.

Intuitively it should be understood that these data objects represent hardware registers, memory units, and buses (hardware data paths) which connect registers to memory units and data transformers. It is assumed that the reader has been introduced previously to the hardware aspects of the QM-1 through readings in [NANO79, SALI76, AGRA76]. Throughout this section it is also assumed that the reader is referring to the pre-defined declarations found in Appendix 2.

5.1 Data Object Declarations

```
<data object> ::= <type decln> | <variable decln> |
                   <pseudovar decln>
```

5.1.1 Type Declaration

```
<type decln> ::= type <type id> {, <type id>} = <type>

<type id> ::= <identifier>

<type> ::= bit | <seq> | <array> | <tuple>

<seq> ::= seq <dimension> bit

<dimension> ::= [ <dec integer> .. <dec integer> ]

<array> ::= array <dimension> of <type id> {<with ptr>} |
            array <dimension> of <seq> {<with ptr>}

<with ptr> := with <var id> {,<var id>}

<tuple> ::= tuple <tuple list> endtup

<tuple list> ::= <var id> : <type list> {<tuple list>}

<type list> ::= <type> | <type id>
```

5.1.1.1 Semantics

Type declarations serve to identify the basic variable classifications (hardware entities) within the QM-1. The most primitive type is a **bit**, representing the binary values 0 and 1. Individual bits may be grouped together into sequences of **bit**'s as in **seq**[5..0]**bit** or **seq**[17..0]**bit**. Intuitively a sequence of bits could represent a hardware register or data path whose value could range from 0 - (2^6-1) or 0 - $(2^{18}-1)$. In S*(QM-1) four classes of hardware registers are declared as **types**: *ls_register*, *es_register*, *f_register* and *k_vector_register* where the first two are represented by 18-bit sequences (**seq**[17..0]**bit**) and the latter by 6-bit sequences.

Two types of memory storage elements are declared--*control_store_word* and *main_store_word*--each of which may contain values in the range 0 - (2^{18}-1). The **type** *bus* represents a uni-directional data path which is used to effect data transfers from a data transformer (shifter, alu, index_alu) or storage unit (main_store, control_store, or external_store) to a variable of **type** *ls_register*. Data on a variable of **type** *bus* may take values in the range 0 - (2^{18}-1) and in the case of a storage unit, will remain stable until a new read operation is performed (external_store is the exception as explained below). Data transformer values will be dealt with in a later section.

Finally two *source* types are declared--*source_all_ones* and *source_all_zeros*--which represent the values 2^{18}-1 and zero respectively.

5.1.2 Variable Declaration

```
<variable decln> ::= var <var id> {,<var id>} :
                          <type list> { : <type list>}

<var id> ::= <identifier> | <identifier> <dimension> |
              <identifier [ <integer> ] |
              <identifier> [ <const id> ] |
              <identifier> [ <var id> ] |
              <identifier> [ <var id> <incrop> ] |
              <var id> . <var id>
```

Variable data objects are declared in terms of previously declared types, either **bit** (the primitive type), **sequences** of bits, **arrays** of types or **tuples**. **Tuples** provide a convenient method of grouping several variables of

possibly different types under one identifier. Variable identifiers may be declared in more than one way in order to clarify the many functions a single hardware register bank or bus may perform. Thus a variable may simultaneously be declared as an **array** and as a **tuple**.

Variables are identified in an S*(QM-1) program in a manner similar to variables in PASCAL programs. Take for example the declaration of local_store (Appendix 2) where local_store[26] and local_store.index[3] refer to the same variable. Where no ambiguities exist the tuple reference may be shortened, as in index[3].

From the point of view of an S*(QM-1) programmer the focus of the QM-1 lies with the variable local_store. Simply put it is an **array** [0..31] of ls_register. As the center of activity all 18-bit data paths lead to or radiate from local_store. These data paths may be individually connected to any of the thirty-two local_store variables under control of the first fourteen f_store **array** elements (known as *residual control*). Each of these f_store variables, i.e. f_store.fmix or f_store.faod, acts as an index into local_store for the particular data path it is associated with. Thus to connect the output of the control_store to local_store[7] we would assign seven to f_store.fcod.

The uni-directional data paths of **type** *bus*: main_store_output, control_store_output, alu_output, index_alu_output and shifter_output are controlled by the f_store variables fmod, fcod, feod, faod, and fsod (with

gspec in this category controlling the index_alu) respectively. Note that in the array declaration of local_store that these variables are used in conjunction with the **with** construct. This indicates that these variables act as indexes into local_store such that data is transferred from a data path to a local_store array element. The data on the buses must be explicitly *gated* into the array element, under programmer control as discussed in sections 7.4.1 and 7.4.2.

Data in a local_store element may similarly be placed on a uni-directional data path acting as an input into memory units, external_store and data transformers. The array elements remain connected to a particular data path until such time as the f_store variable acting as a the index pointer is altered.

The variable local_store is also declared as a **tuple**, the purpose of which is twofold. Firstly it indicates that four variables, index[0] to index[3] (local_store[24] to local_store[27]) are associated via f_store.fmpc to the microprogram counter incrementing unit. Secondly it indicates that local_store[31], instruction_reg, is of type ls_register but at the same time consists of three 6-bit registers (c, a, b) and an opcode followed by two parameters. The former serves to indicate that local_store[31] is also a gateway into the 6-bit domain of f_register and k_vector_register, and in the latter that a relationship exists between instructions in control_store

and local_store[31] (to be discussed later).

The variable external_store serves several purposes denoted by the tuple declaration. The first eight elements are port registers for communication with the external environment. The next eight elements act as the right inputs into the index alu, which is indicated in the declaration of the variable index_alu_x (right input). The remaining elements act as base address and field lengths for mainstore protection and as interrupt addresses and hardware interrupt enable bits.

Note that input data to external_store is supplied from a local_store element denoted by the contents in f_store.feid. Since the range of feid is 0 - (2^6-1) and there are only thirty-two local_store elements the input data bus will be connected to a source of all ones in the cases where the value in feid is greater than thirty-one. This is a common occurrence for a residual control f_store variable which control data paths using a local_store variable as input. In some instances the data path is set to zero.

Variables of **type** *k_vector_register* may be partitioned into three groups. The first, ka and kb, are general purpose 6-bit variables providing temporary storage, while the second, kx, ks and kt, act as 6-bit masks in conjunction with test expressions (section 7.4.3). The third group consists of kalc, ksha, and kshc which control the operation of the ALU and shifter (section 7.4.2). When not serving

their primary function these 6-bit variables may be used for temporary storage.

5.1.3 Pseudo Variable Declaration

```
<pseudovar decln> ::= pvar <identifier>
                          {,<identifier>} : <type list>
```

5.1.3.1 Semantics

Pseudo variables are declared in terms of **bits** or **sequences** of bits. They are abstractions of control fields within a nanoword and as such are not true variables, except for the **pvar** *sw* which represents a 6-bit data input from the front panel switches. They may only be used as the left-hand side variable in a transfer statement (except for sw which may only be an assignment source) and may only have constants or integers assigned to them, which may only have the decimal equivalents of 0, 1, 2 or 3. Since these **pvars** represent fields within a nanoword no action is taken if the assignment is to zero, except in the case of alu_carry_out and alu_carry_in. The actions associated with each **pvar** is as follows:

1. carry_control := 0 -- no-op

   carry_control := 1 -- ALU carry transferred to COH

   carry_control := 2 -- shifter end bit transferred to COH

   carry_control := 3 -- ALU carry transferred to COH and CIH

2. Alu_carry_out and alu_carry_in serve to set the COH and

CIH to one or zero.

3. Aux_action executes the auxiliary action identified with the variable f_store.fact.
4. The rmi represents the rotate/mask/index unit used with the main_store_output and is always set to zero (bypass) unless explicitly set by the programmer. 1--use parameter set a, 2--use parameter set b, and 3--use parameter set c.
5. Load_npc := 1 causes the seven high order bits on the control_store_output variable (bus) to be transferred to the NPC (nanoprogram counter) for use in instruction procedure invocation. The low order eleven bits are saved in a dedicated register for use with load_r31.
6. Load_r31 := 1 causes the eleven bits in the dedicated register last saved using the load_npc pvar to be transferred to the low order eleven bits of instruction_reg (a_parameter and b_parameter), and sets instruction_reg.opcode to zero.

5.2 Synonym Declaration

<syn decln> ::= syn <identifier> = <var id>

5.2.1 Semantics

Synonym declarations are used to rename pre-defined data objects to aid in the textual clarity of an S*(QM-1) program. When declared in the declaration block they are global in scope remaining in existence until the completion

of a program. For example:

```
        syn alu_operation = kalc
        syn pdp11-register = local_store[0..7]
```

When using an array bound with the variable identifier the lower bound must be zero.

5.3 Constant Declaration

```
<const decln> ::= const <const id> : <radix>
                       (6) <digit> {<digit>}

<const id> ::= <identifier>
```

5.3.1 Semantics

Constant declarations provide a convenient method of identifying inter values. The number of binary digits associated with the constant is denoted by (**<digit>**) where the digit must be either 6, 2, or 1. For example:

```
        const double_left : bin (6) 1100
        const new_mode : dec (6) 4
```

New_mode would thus be expanded by the compiler to the binary value 000100.

5.4 Case Declaration

```
<case decln> := case <identifier> : <integer>
```

5.4.1 Semantics

A case declaration indicates to the compiler the number of position dependent branch statements which will be

associated with the case statement identified by <identifier>. All case statements must be identified in the declaration section in this manner. For example:

```
        case jump_table : 8
```

6. Initialization Block

```
<initialization block> ::= <empty> |
                           init <init list> endinit

<init list> ::= <var id> := <integer> {,<integer>}
                    {<init list>}
```

The initialization block provides a means for initializing register variables before execution of the program begins. The variable id may include local, external or 'f' store register variables (or their global synonyms) where the integer value should not exceed 63 (2^6-1) for f_store and $2^{18}-1$ for local and external store. Thus to establish the microprogram counter and initialize it to oct 300 use:

```
                fmpc := 25
                local_store[25] := oct 300
```

7. Execution Block

```
<execution block> ::= <procedure> {<procedure>}
```

7.1 Procedure Declaration

```
<procedure> ::= proc <proc id> (<proc param>) <syn block>
                 <exec stmt> {<exec stmt>} endproc

<proc id> ::= <identifier>

<proc param> ::= instruction <inst param> |
                  interrupt < interrupt param > |
                  subroutine <subroutine param>
```

7.1.1 Semantics

The execution block of an S*(QM-1) program consists solely of procedures. Each procedure is identified by a unique name <proc id> and has associated with it one or more parameters. The first parameter indicates the type of procedure, namely: **instruction**, **subroutine**, or **interrupt**.

Instruction procedures form interpreters for control store 18-bit instructions in which the high order seven bits form the opcode. The opcode associated with an instruction procedure is dependent upon is position in a program relative to other instruction procedures. The first instruction procedure is assigned the opcode zero, the second one, and so on. The number of instruction procedures in a program is limited to 128 (2^7).

To invoke the next control store instruction, which has been pre-fetched and is on the control_store_output *bus* the sequence:

```
        local_npc := 1;
        local_r31 := 1;
```

must be issued. Normally these assignments will form the last statements within the procedure. On completion of the last statement in the procedure interrupts are permitted to occur. If no interrupts are taken the next instruction begins execution.

Subroutine procedures provide support for instructions. They may only be called from instruction procedures and return explicitly via the **return** statement or implicitly after the last statement in the subroutine is executed. Subroutines may also be **activated** (section 7.4.4) from an instruction with the understanding that the subroutine must cause the next instruction to be invoked. Interrupts will be allowed after the last statement if the parameter *allow_ints* has been specified. Due to the QM-1 hardware subroutines are severly limited in their use of test statements (section 7.4.3). They may only employ a **return** or **cocycle** statement as the body of the test statement.

Interrupt procedures form *handlers* for one or more of the 29 external interrupts which are possible on the QM-1. Each interrupt has a parameter identifying the hardware interrupt(s) which they are to service. The limitation imposed on subroutines also apply to interrupts. Since interrupts are serviced prior to commencement of the next instruction the use of the **return** statement will cause control to be transferred to the first statements of the

next instruction.

7.2 Local Synonym Declaration

```
<syn block> ::= <empty> | <syn decln> {<syn decln>}
```

7.2 Local Synonym Declaration

Local synonyms identify a global synonym or pre-defined variable for use within that procedure. The synonym is recognized within the procedure in which it was declared. Thus, for example, synonyms do not extend to subroutines which are called or activated from an instruction procedure.

7.3 Executable Statements

```
<exec stmt> ::= <simple stmt> ; | <label stmt> |
                <case stmt> ; | <parallel stmt>|
                <composite stmt>
```

7.4 Simple Statements

```
<simple stmt> ::= <transfer stmt> | <function stmt> |
                  <test stmt> | <branch stmt> |
                  <return stmt>
```

7.4.1 Transfer Statement

```
<transfer stmt> ::= <var id> {,<var id>} := <source>

<source> ::= <var id> | <const id> | <integer>
```

7.4.1.1 Semantics

A transfer statement corresponds to the situation where data is transferred directly from the source to the variable with no intervening data transformations being performed. The following rules apply:

1. Constants and integers, which must be less than 2^6 (64), may only be assigned to variables of **type** *k_vector_register* or *f_register*. **Pvar**'s may only be assigned values of 0, 1, 2, or 3 depending upon their declaration as described in section 5.1.3.1. For example:

    ```
    f_store.feid := oct 54;
    ka := 3;
    ```

2. 6-bit variables are grouped into the following five sets:

    ```
    1 = { sw }
    2 = { instruction_reg.c, a, b, ka, kb, kt, kx }
    3 = { f_store[0..31] }
    4 = { c, a, b, ka, kb, kalc, ksha, kshc }
    5 = { f_store.g_store[0..11] }
    ```

    Variables of sets one and two may be used as sources for transfers to variables of set three. Variables of sets one, two and three may be used as sources for transfers to set four. Variables of set five may be used as sources for transfers to sets three or four.

3. 18-bit transfers must include at least one reference to either a local_store or an external_store variable. The action taken by the compiler is to set a residual

control register (f_store variable) and issue a *gate nanoprimitive*, if required. For example the statement:

```
        local_store[10] := external_store[9];
```

is the high level equivalent of the following sequence of legal lower level statements:

```
        f_store.feod := 10;
        f_store.feoa := 9;
        local_store[feod] := external_store[feoa];
```

where in the final statement the ':=' serves to indicate to the compiler that only the *gate es* nanoprimitive need by issued. The basic rule is that if an integer or constant index is specified the associated f_store variable (there is only one) will always be assigned the integer index first (residual control setup). If the f_store variable is used as the index, i.e., local_store[fail], the setup need not be performed as the programmer has indicated that it has been assigned the appropriate value previously.

A statement of the type:

```
        alu_input_right := local_store[fail];
```

is a null statement as the data from the local_store element is already assigned to the bus by virtue of the fact that f_store.fail was previously setup.

A close perusal of the pre-defined declarations in regards to **array** ... **with** pointers will give the correct residual control index for the application required.

7.4.2 Function Statement

```
<function stmt> ::= <var id> := <expression>

<expression> ::= <alu const> | <alu unary op> <var id> |
                 <var id> <incr op> |
                 <shift op> <shift count> <var id> |
                 <var id> <alu op> <var id> |
                 <var id> <alu op> <const id>

<alu const> ::= ↓x↓ones | ↓x↓zero

<alu unary op> ::= <const id> | ↓x↓notl | ↓x↓notr |
                   ↓x↓passl | ↓x↓passr | ↓x↓incl |
                   ↓x↓incr

<incr op> ::= +1 | +2 | -1 | +b | +ab

<shift op> ::= a<<↓d↓ | a>>↓d↓ | c<<↓d↓ | c>>↓d↓ |
               l<<↓d↓ | l>>↓d↓ | <const id> |
               (<var id>)

<shift count> ::= <integer> | <const id> | (<var id>)

<alu op> ::= + | - | and | nand | or | nor |
             xor | <const id> | (<var id>)
```

7.4.2.1 Semantics

Expressions represent data transformations of at most two terms and are limited to operations which can be performed in one pass through an ALU or shifter. Alu operations using a constant expression (**ones**, **zero**) regard the right and left inputs in a *don't care* manner as their values do not affect the value placed on the output. Similarly unary operators transform only one input, such that the other input is a *don't care* condition. Four types of data transformations will be examined.

1. The basic format of an ALU functions statement is:

```
local_store[faod] := local_store[fair] (Kalc)
```

```
            local_store[fair];
```

which indicates that all residual control f_store variables and the ALU operation specified in the k_vector_register variable *kalc* have been previously set. Only the transfer of data from the variable alu_output need be performed at this stage. If variable alu_output is substituted for local_store[faod] no transfer is performed as the results of alu operations are automatically placed on the output *bus*.

The right hand side of the statement may be replaced by higher level expressions to provide a higher level of abstraction, but these are always encoded by the compiler so that the above statement is achieved. Only a limited number of ALU operations are directly supported from the total of sixteen arithmetic and sixteen logical operations. If a non-supported operation is required *kalc* must be setup before the ALU operation is performed. For example:

```
    kalc := bin 011101; /* left and not right */
    local_store[5] := local_store[10] (kalc)
            local_store[11];
```

When this method is employed the assignment to *kalc* must precede the operation each time it is used as the k_vector_register is volatile and no guarantee may be made as to its contents after an alu operation.

Alu constants (without the preceding 'x') are used to assign either zero or $2^{18}-1$ to the output variable.

When using unary operators the correct f_store residual control variable must be used. For example:

```
    local_store[10] := notr local_store[fair];
    local_store[faod] := passl local_store[fair];
           /* incorrect specification */
```

The use of fair in the latter statement is semantically incorrect because it is incompatible with the ALU operation. **Passl** and **passr** serve to transmit either the left input or right input directly through the ALU without altering their values.

Alu binary operators (+, -, **and**, **nand**, **or**, **nor**, **xor**) are provided for convenience as they represent the binary operator used at least 80% of the time. For example:

```
    local_store[10] := local_store[5] nand
           local_store[6];
```

is equivalent to:

```
        faod := 10;
        fail := 5;
        fair := 6;
        kalc := bin 010001; /* nand */
        local_store[faod] := local_store[fail]
           (kalc) local_store[fair];
```

Four local condition bits--result, carry, sign and overflow--are generated with each ALU operation. These conditions may be transferred to the 6-bit variable f_store.fist for later evaluation using the **pvar**

alu_status and the assignment: alu_status := 1;. This assignment is always associated with the previous ALU operation.

2. Index ALU transformations require several levels of indirection to specify the inputs and operation to be performed. The left input is a local_store array element which must not include index[0..3]. It is determined by the contents of one of the following variables: a, b, kx, ka, kb or gspec (where gspec represents backup[0..11], b, ksha, kx or ks). The local_store index must be one of these variables; or an integer or constant in which case one of kx, ka or kb will be used to encode the index value. The right input consists of index_alu_x variables and associated indexes. The output of the index ALU is encoded in one of the gspec variables.

Thirteen operations are directly available as denoted by an operator preceded by 'x' and (**and**, **xor**, **or**, -, +). The 'x' is required for unary operations to avoid any ambiguity between the ALU and index ALU. In addition the operation, one of sixteen logical and sixteen arithmetic transformations, may be encoded into one of the variables: a, b, ka, kb, fmpc, fidx, and backup[0..11]. For example:

```
local_store[gspec.b] := local_store[a] (backup[3])
                    index_alu_x[g_input[31]];
    local_store[5] := local_store[10] and
```

```
            operand_source[3];
```

where the indexes will be correctly assigned to variables of the type k_vector_register.

3. The operators (+1, +2, +b, +ab) are used in conjunction with the MPC incrementing unit. The basic format of MPC operations is:

```
    local_store.index[fmpc] := index[fmpc] +b;
```

where the variable on the left and right hand sides of the assignment must be the same. Thus:

```
    index[3] := index[2] +ab; /* incorrect */
```

is semantically incorrect.

When performing control_store read operations the MPC may be used to provide increment values for use as the address. A typical read operation would be:

```
local_store[fcod] := control_store[index[fmpc] +2 ];
```

which would increment the local_store register pointed to by fmpc (modulo 4) by two and use this as the address into control_store. The word read is assigned to the local_store element indexed by fcod.

All the variables f_store[0..31] (except for f_store.fiph) may be incremented, +1, or decremented, -1. Both variables in this type of function statement must be the same. For example:

```
    f_store.fmix := f_store.fmix -1;
    fair := fair +1;
```

4. Aluf binary expressions consist of a left and right

input made up from sets one, two and three (section 7.4.1.1.(2)) where at most one term from sets one and two may be used. The variable assigned to must be from set three.

The ALUF operation is encoded into one of the following variables: a, b, kt, kb and g_input[0..4]. If the operator is directly specified it will be encoded into one of the variables kt or kb. Since ALUF operations may easily be distinguished from ALU operations the same operator are used namely: **notl**, **nor**, **nand**, **zero**, **one**, xor, **passr**, **+**, **and**, **passl**, **or**, **-**. Four operators are not directly supported and must be encoded by the programmer. For example:

```
fmix := fmix + fmix;
fair := zero;
fmod := fidx (b) fmpc;
```

Constants and integers may be used as one of the variables, in which case the compiler encodes the integer or constants into one of the 6-bit k_vector_register variables: ka, kb, kt or kx. For example:

```
fmix := fmix + oct 12;
fmpx := 3 + fcod;
```

5. Shifter operations are controlled by two variables *kshc* (shifter control) and *ksha* (amount of shift) in the same manner that *kalc* controls the operation of the ALU. The

most basic statement is therefore:

local_store[fsod] := (kshc) (ksha) local_store[fsid];

Operators have been supplied to support circular (c), arithmetic (a), and logical (l) shifts. Each of these shifts may be in a right (>>), or left (<<) direction. The optional modifier 'd' specifies a double shift (36-bits) using the shifter extension (18-bits from alu_output) as the high order 18-bits. The shift count (amount the input is to be shifted) is specified following the type of shift. For example:

```
local_store[5] := c>>3 local_store[7];
```

which performs a right circular shift of three positions on the contents of local_store[7] and assigns the output to local_store[5];.

Two local condition bits shb (shifter high bit) and slb (shifter low bit) may by transferred to f_store.fist for global testing using the assignment:

```
shifter_status :=1;.
```

The use of this assignment always applies to the preceding shift statement.

7.4.3 Test Statement

```
<test stmt> ::= if (<test expression>) <compound list> fi

<test expression> ::= <test modifier> <condition test>
                      <condition op> <digit> |
                      <condition test> <condition op>
                          <digit> |
                      <var id> <condition op> <var id> |
                      ( <expression> ) <condition op>
```

```
                         <digit>
<test modifier> ::= special | local | global

<condition test> ::= (<var id>) | <const id> |
                     overflow | carry | shb |
                     slb | ms_busy | ms_data |
                     r_index | sign | result

<condition op> ::= <const id> | > | !> | < |
                   !< | == | !=
```

7.4.3.1 Semantics

Tests are conducted on one of three sets of conditions:

1. Current conditions available from the shifter and ALU (**shb**, **carry**, **sign**, **result**, **overflow** and **slb**). **Shb** represents the higher order bit in the variable shifter_output and **slb** represents the low order bit on shifter_output.
2. Global conditions previously saved in f_store.fist (same as above).
3. Special machine status conditions (**r_index**, **ms_busy**, and **ms_data**). **R_index** is true if the result of the last index_alu operation involving an assignment to a local_store variable was *not zero*. **Ms_busy** is true if a main_store read/write operation is in progress, and ms_data is true if data pertaining to the last read operation is not yet available on the main_store_output.

A test modifier--**local**, **global** or **special**--is used to specify which set of conditions is to be tested. This modifier is optional and if any ambiguity exists (between **local** and **global**) **local** conditions will be tested. Three

6-bit k_vector_registers: kt (**local**), ks (**global**) and kx (**special**) act as masks for the conditions specified. If a combination of several bits is needed to test a condition the appropriate mask should be assigned the valued before the test expression is specified.

Only the digits zero or one are permitted in tests, such that a bit in a condition is tested for zero or one. Two local_store variables may be tested, using the ALU, for equality: == (equal) or != (not equal); or inequality: >, <, !>, !<. **Local** conditions are tested with the result of the operation being placed on alu_output. An expression involving unary or binary operations employing the ALU may also be used. Finally f_store variables may be tested against zero: !=0 or ==0.

Several restrictions are placed on the contents of the compound list by the type of procedure in which the statement is used. The following rules apply:

1. In **instruction** procedures the statement may be any of the statements listed in <compound list>. If the first statement is a **cocycle** statement it is assumed to be the only statement within the test statement and is encoded by the use of the *skip nanoprimitive*. If several statements beginning with a **cocycle** are required they must be enclosed in a **begin**....**end** construct.
2. In **subroutine** and **interrupt** procedures the statements within the test statement may consist only of a **return** statement or a **cocycle** statement.

7.4.4 Branch Statement

```
<branch stmt> ::= goto <branch label> |
                   call <branch label> |
                   act  <branch label>.

<branch label> ::= <label id> | <proc id>
```

7.4.4.1 Semantics

The semantics regarding branch statements is straightforward:

1. Only an **instruction** procedure may **call** another procedure and it may only **call subroutines**. The return will be made to the statement following the **call** statement. A subroutine may explicitly return using the **return** statement or it may implicitly return on completion of the last statement in the subroutine.
2. A **goto** statement transfers control to the statement following a label statement identified by <label id>. It may be employed in any of the three types of procedures. Transfers may only be effected to **label** statements in the same procedure as the **goto**.
3. The **act** branch statement transfers control to a procedure with the understanding that control will not be returned to the activating procedure. Any of the three types of procedures may be activated from any other procedure.

   It is semantically incorrect for an activated **subroutine** procedure to use a **return** statement as no effort is made to save a return address. If an **act**

branch statement is the last statement in an **instruction** procedure interrupts will not be taken.

7.4.5 Return Statement

<return stmt> ::= **return**

7.4.5.1 Semantics

The **return** statement may only be used in a subroutine procedure to effect a transfer of control to the statement following the calling statement. A **return** statement must not be used in any subroutine which is **act**ivated by another procedure.

7.5 Label Statement

<label stmt> ::= **label** : <label id> (<label param>)

<label id> ::= <identifier>

7.5.1 Semantics

Label statements serve to identify, by name, a position in a procedure which is a target of a goto statement. The <label param> is an identifier which is used to associate the **label** with a specific **case** statement, and is optional if it is not the target within a **case** statement.

7.6 Case Statement

```
<case stmt> ::= case <case id> of <case list>
                <case transfer > endcase

<case id> ::= <identifier>

<case list> ::= <integer> : <branch stmt> {<integer>
                : <branch stmt>}

<case transfer> ::= <empty> | <transfer stmt>
```

7.6.1 Semantics

Case statements make use of special hardware features within the QM-1 nano-architecture to effect multiway branches. Each **case** statement must have been previously identified in the declaration block (section 5.4). The body of a **case** statement consists of **branch** statements of the form:

goto <label id>

or

act <proc id>

These statements are order dependent in that they must follow the order in which they appear in the program. Each statement is identified with an integer which must begin with zero and must be consecutive and may not exceed the declared number of statements (section 5.4.1).

The variable local_store.instruction_reg.b is used to hold an index value representing the **branch** statement (as identified by its integer) and must have been previously assigned before the case statement is executed.

Case statements are guaranteed to alter the variable local_store.instruction_reg during the initial stage of execution so provision has been made (<case transfer>) to assign, via a transfer statement, a value to this variable. For example:

```
instruction_reg.b := 3;
faod := 31;
case jump_table of
    0 : goto transfer1 .
    1 : goto transfer2
    2 : goto transfer3
    3 : goto transfer4
    local_store[faod] := alu_output
endcase ;
......
label : transfer3 (jump_table) S1;
```

This will cause control to be transferred to the statement S1, with local_store[31] containing the value which was present on the alu_output bus. The convention is that the transfer statement must be capable of execution within a **cocycle** statement.

Instruction procedures may not be the target of an **act** statement within **case** statements.

7.7 Parallel Statement

<parallel stmt> ::= <cocycle stmt> ;

```
<cocycle stmt> ::= cocycle <cocycle list> coend

<cocycle list> ::= <simple stmt> □ <simple stmt> |
                     <cocycle list> □ <simple stmt>
```

7.7.1 Semantics

The **cocycle** construct specifies that the simple statements (not including <test stmt>'s) separated by '□' are to be executed within the same *T-step* (a single 72-bit *T-vector* with its associated *k-vector*).

Thus, semantically correct individual simple statements may be specified in a **cocycle** statement such that the result will not be executable in a single T-step due to timing or resource conflicts. We call this type of error a *mapping error*.

The compiler (compactor) will inform the programmer when a mapping error occurs but is unable to supply any additional information.

Example:

```
cocycle
    f_store.fist := fail + fair
    □ local_store[faod] := ones
    □ local_store[fmod] := main_store_output
coend;
```

7.8 Composite Statements

```
<composite stmt> ::= <compound stmt> ; | <region stmt> ; |
                       <repetition stmt> ;
```

7.8.1 Compound Statement

```
<compound stmt> ::= begin <compound list> end

<compound list> ::= <simple stmt>  | <cocycle stmt> |
                     <region stmt> | <compound stmt> |
                     <compound list> ; <simple stmt> |
                     <compound list> ; <cocycle stmt> |
                     <compound list> ; <region stmt> |
                     <compound list> ; <compound stmt> |
```

7.8.2 Region Statement

```
<region stmt> ::= region <compound list> endreg
```

7.8.2.1 Semantics

Statements within a **region** statement, i.e. simple, cocycle and other region statements are guaranteed to execute in strict sequential order. For example:

```
        region
            S1 ; S2 ;
            cocycle S3 □ S4 coend ;
            S5
        coend ;
```

specifies that S2 will not begin executing before S1 has completed execution. Similarly S5 will not begin until S3 and S4 (executed in parallel) have completed.

7.8.3 Repetition Statement

```
<repetition stmt> ::= repeat <compound list> until
                        (<test expression>) |
                        while (<test expression>) do
                        <compound list> endwhile
```

7.8.3 Semantics

The **repeat** statement specifies that the statements within <compound list> are executed until <test expression> is true. <compound list> is always executed at least once. The statements within <compound list> must be mappable into a single nanoword.

Once again we have the situation where a semantically correct statement is still unexecutable due to the compiler's (compactor) inability to map the statements within <compound list> into a single nanoword.

The **while** statement is equivalent to:

```
label : start
if (<test expression>)
    <compound list>
    goto start
fi;
```

Thus, <compound list> will not be executed when it is first encountered unless (<test expression>) is true. Subsequently <compound list> will be executed until (<test expression>) becomes false.

APPENDIX 2. S*(QM-1) PREDEFINED DATA OBJECTS

PREDEFINED DECLARATIONS FOR S*(QM-1)

TYPE DECLARATIONS

```
type ls_register            = seq [ 17..0 ] bit
type es_register            = seq [ 17..0 ] bit
type main_store_word        = seq [ 17..0 ] bit
type control_store_word     = seq [ 17..0 ] bit
type source_all_ones        = seq [ 17..0 ] bit
type source_all_zeros       = seq [ 17..0 ] bit
type bus                    = seq [ 17..0 ] bit
type f_register             = seq [ 5..0 ] bit
type k_vector_register      = seq [ 5..0 ] bit
```

Pseudo Variable Declarations

```
pvar alu_carry_out, alu_carry_in : bit
pvar carry_control : seq [ 1..0 ] bit
pvar alu_status, shifter_status : bit
pvar aux_action : bit
pvar rmi_select : seq [ 1..0 ] bit
pvar load_npc : bit
pvar load_r31 : bit
pvar external_interface_control : bit
pvar external_interface_input   : bit
pvar sw : seq [ 5..0 ] bit
```

Variable Declarations

```
var main_store_output, external_store_output      : bus
var control_store_output                          : bus
var alu_output, shifter_output, index_alu_output : bus

var ka, kb, ks, kx, kt : k_vector_register
var kalc, ksha, kshc   : k_vector_register
```

```
var local_store
        : array [ 0..31 ] of ls_register
              with fmod, fcod, faod, fsod, feod, gspec
        : tuple
                general_purpose   : array [ 0..23 ] of ls_register
                index                  : array [ 0..3 ] of ls_register
                                         with fmpc
                general_purpose2 : array [ 0..2 ] of ls_register
                instruction_reg   :  ls_register
                                  :  tuple
                                           c : f_register
                                           a : f_register
                                           b : f_register
                                      endtup
                                  :   tuple
                                           opcode      : seq [ 6..0 ] bit
                                           a_parameter : seq [ 4..0 ] bit
                                           b_parameter : seq [ 5..0 ] bit
                                      endtup
          endtup

var external_store
        : array [ 0..31 ] of es_register
              with feia
        : tuple
               port_register      : array [ 0..7 ] of es_register
               operand_source     : array [ 0..7 ] of es_register
               base_address       : es_register
               field_length       : es_register
               interrupt_enable   : array [ 0..1 ] of es_register
               alt_base_address   : es_register
               alt_field_length   : es_register
               interrupt_address  : array [ 0..9 ] of es_register
          endtup

var external_store_input_data
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with feid
        : tuple
                local_store     : array [ 0..31 ] of ls_register
                all_ones        : array [ 0..31 ] of source_all_ones
          endtup

var external_store_output_address
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with feoa
        : tuple
                external_store    : array [ 0..31 ] of es_register
                all_zeros         : array [ 0..31 ] of source_all_zeros
```

```
        endtup

var f_store                                    /* residual control f registers        */
        : array [ 0..31 ] of f_register
        : tuple
              fmix   : f_register          /* main store input--address/data      */
              fmod   : f_register          /* main store output--data             */
              fcia   : f_register          /* control store input--address        */
              fail   : f_register          /* alu input--left                     */
              fcid   : f_register          /* control store input--data           */
              fair   : f_register          /* alu input--right                    */
              fcod   : f_register          /* control store output--data          */
              faod   : f_register          /* alu output--data                    */
              fsid   : f_register          /* shifter input--data                 */
              fsod   : f_register          /* shifter output--data                */
              feid   : f_register          /* external store input--data          */
              feod   : f_register          /* external store output--data         */
              feia   : f_register          /* external store input--address       */
              feoa   : f_register          /* external store output--data         */
              fact   : f_register          /* auxilary action */
              fuser  : f_register          /* user partition  */
              fmpc   : f_register          /* mpc pointer     */
              fidx   : f_register          /* index           */
              fist   : f_register          /* global status   */
              fiph   : f_register          /* phantom         */
              g_store : array [ 0..11 ] of f_register
        endtup

var control_store : array [ 0..40959 ] of control_store_word
                        with control_store_address_source

var control_store_address_source
        : tuple
          register_address     : array [ 0..63 ] of seq [ 17..0 ] bit
                                      with fcia
                               : tuple
                                      local_store     : array [ 0..31 ] of ls_register
                                      all_ones        : array [ 0..31 ] of source_all_ones
                                 endtup
          control_store_output : bus
          index                  : array [ 0..3 ] of ls_register
                                      with fmpc
          index_alu_output     : bus
        endtup

var control_store_data
        : array [ 0..63 ] of seq [ 17..0 ] bit
             with fcid
        : tuple
```

```
                local_store     : array [ 0..31 ] of ls_register
                all_ones        : array [ 0..31 ] of source_all_ones
          endtup

  var main_store : array [ 0..98303 ] of main_store_word
                        with main_store_source

 var main_store_destination
        : array [ 0..39 ] of seq [ 17..0 ] bit
              with fmod     /* if fmod > 39 then null operation results */
        : tuple
                local_store    : array [ 0..31 ] of ls_register
                port_register  : array [ 0..7 ] of es_register
           endtup

 var main_store_source
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with fmix
        : tuple
                local_store    : array [ 0..31 ] of ls_register
                port_register  : array [ 0..7 ] of es_register
                all_ones       : array [ 0..23 ] of source_all_ones
           endtup

  var gspec
        : array [ 0..15 ] of seq [ 5..0 ] bit
        : tuple
                back_up : array [ 0..11 ] of f_register
                ksha    : k_vector_register
                b       : f_register
                ks      : k_vector_register
                kx      : k_vector_register
          endtup

 var alu_input_left
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with fail
        : tuple
                local_store     : array [ 0..31 ] of ls_register
                all_ones        : array [ 0..31 ] of source_all_ones
          endtup
```

```
var alu_input_right
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with fair
        : tuple
                local_store     : array [ 0..31 ] of ls_register
                all_ones        : array [ 0..31 ] of source_all_ones
          endtup

var shifter_input
        : array [ 0..63 ] of seq [ 17..0 ] bit
              with fsid
        : tuple
                local_store     : array [ 0..31 ] of ls_register
                all_ones        : array [ 0..31 ] of source_all_ones
          endtup

 var index_alu_x
        : array [ 0..11 ] of seq [ 17..0 ] bit
              with alu_indirect_specifier
        : tuple
                operand_source        : array [ 0..7 ] of es_register
                x_all_ones            : seq [ 17..0 ] bit
                regs                  : seq [ 17..0 ] bit
                main_store_output     : bus
                control_store_output  : bus
          endtup

var alu_indirect_specifier
        : array [ 0..7 ] of seq [ 5..0 ] bit
        : tuple
                a       : f_register
                b       : f_register
                kt      : k_vector_register
                kb      : k_vector_register
                g_input : array [ 28..31 ] of f_register
           endtup
```

APPENDIX 3. A SHORT PROGRAMMING EXAMPLE

In the example shown below, we describe a control store instruction procedure which adds the contents of the local store register specified by the first parameter in the instruction to the immediate value following it (already prefetched and on the control store output data bus), and uses this as an address into control store. The 18-bit value read from control store is placed into the local store register (using residual control register fcod as a pointer) specified by the second parameter. The microprogram counter, indicated by the value in residual control store register fmpc, is then incremented by 2 and a fetch subroutine is activated. Within a microprogram, the actual instruction would take the form: "arel 4,5,17703". Note the use of synonyms within the declaration block.

```
prog (test)

    declaration

         /* included files expanded by preprocessor */
        #include "qm1_dec.h"

        syn immediate = control_store_output
        syn index_adr = index_alu_output
        syn base_register = a_parameter
        syn data_register = b_parameter

        macro INCR_MPC_2
             index[ fmpc ] := index[ fmpc ] +2
        endmacro

    endec

    init
            /*
               local store register 24 set
               as the microprogram counter
            */
        fmpc := 24
    endinit
```

```
    proc add_relative(instruction, op=arel, fmt=r.r.c)

        fcod := data_register;

        region

            index_alu_output := local_store[ base_register ]
                                    + immediate;

            local_store[fcod] := control_store[index_adr];

        endreg

        INCR_MPC_2;

        act fetch;

    endproc
         ...
         ...
endprog
```

Note also, that the index ALU is used since one of its inputs may be the data on the control store output data bus. The result of the index ALU operation is left on the index ALU bus after which it is used as an address into control store. The **region** construct is used to ensure that data dependency conflicts do not arise.